In [ ]:

In [ ]:

Import The Relevant Libraries

In [1]:
```python
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
```

Extract the data from CSV

In [3]:
```python
# Load the data
raw_csv_data = np.loadtxt('D:\Projects\python\Audiobooks\Audiobooks_data.csv', delimiter = ',')

# The inputs are all columns in the csv, except for the first one [:,0]
# (which is just the arbitrary customer IDs that bear no useful information),
# and the last one [:,-1] (which is our targets)

unscaled_inputs_all = raw_csv_data[:,1:-1]

# The targets are in the last column. That's how datasets are conventionally organized.
targets_all = raw_csv_data[:,-1]
```

```
<>:2: SyntaxWarning: invalid escape sequence '\P'
<>:2: SyntaxWarning: invalid escape sequence '\P'
C:\Users\fady\AppData\Local\Temp\ipykernel_18880\417664507.py:2: SyntaxWarning: invalid escape sequence '\P'
  raw_csv_data = np.loadtxt('D:\Projects\python\Audiobooks\Audiobooks_data.csv', delimiter = ',')
```

Balance the dataset

In [4]:
```python
# Count how many targets are 1 (meaning that the customer did convert)
num_one_targets = int(np.sum(targets_all))

# Set a counter for targets that are 0 (meaning that the customer did not convert)
zero_targets_counter = 0

# We want to create a "balanced" dataset, so we will have to remove some input/target pairs.
# Declare a variable that will do that:
indices_to_remove = []

# Count the number of targets that are 0.
# Once there are as many 0s as 1s, mark entries where the target is 0.
for i in range(targets_all.shape[0]):
    if targets_all[i] == 0:
        zero_targets_counter += 1
        if zero_targets_counter > num_one_targets:
            indices_to_remove.append(i)

# Create two new variables, one that will contain the inputs, and one that will contain the targets.
# We delete all indices that we marked "to remove" in the loop above.
unscaled_inputs_equal_priors = np.delete(unscaled_inputs_all, indices_to_remove, axis=0)
targets_equal_priors = np.delete(targets_all, indices_to_remove, axis=0)
```

Standardize the inputs

In [5]:
```python
# That's the only place we use sklearn functionality. We will take advantage of its preprocessing capabilities
# It's a simple line of code, which standardizes the inputs, as we explained in one of the lectures.
# At the end of the business case, you can try to run the algorithm WITHOUT this line of code.
# The result will be interesting.
scaled_inputs = preprocessing.scale(unscaled_inputs_equal_priors)
```

Shuffle the data

In [6]:
```python
# When the data was collected it was actually arranged by date
# Shuffle the indices of the data, so the data is not arranged in any way when we feed it.
# Since we will be batching, we want the data to be as randomly spread out as possible
shuffled_indices = np.arange(scaled_inputs.shape[0])
np.random.shuffle(shuffled_indices)

# Use the shuffled indices to shuffle the inputs and targets.
shuffled_inputs = scaled_inputs[shuffled_indices]
shuffled_targets = targets_equal_priors[shuffled_indices]
```

Split the dataset into Train, Validation, and Test

In [7]:
```python
# Count the total number of samples
samples_count = shuffled_inputs.shape[0]

# Count the samples in each subset, assuming we want 80-10-10 distribution of training, validation, and test.
# Naturally, the numbers are integers.
train_samples_count = int(0.8 * samples_count)
validation_samples_count = int(0.1 * samples_count)

# The 'test' dataset contains all remaining data.
test_samples_count = samples_count - train_samples_count - validation_samples_count

# Create variables that record the inputs and targets for training
# In our shuffled dataset, they are the first "train_samples_count" observations
train_inputs = shuffled_inputs[:train_samples_count]
train_targets = shuffled_targets[:train_samples_count]

# Create variables that record the inputs and targets for validation.
# They are the next "validation_samples_count" observations, folllowing the "train_samples_count" we already assigned
validation_inputs = shuffled_inputs[train_samples_count:train_samples_count+validation_samples_count]
validation_targets = shuffled_targets[train_samples_count:train_samples_count+validation_samples_count]

# Create variables that record the inputs and targets for test.
# They are everything that is remaining.
test_inputs = shuffled_inputs[train_samples_count+validation_samples_count:]
test_targets = shuffled_targets[train_samples_count+validation_samples_count:]

# We balanced our dataset to be 50-50 (for targets 0 and 1), but the training, validation, and test were
# taken from a shuffled dataset. Check if they are balanced, too. Note that each time you rerun this code,
# you will get different values, as each time they are shuffled randomly.
# Normally you preprocess ONCE, so you need not rerun this code once it is done.
# If you rerun this whole sheet, the npzs will be overwritten with your newly preprocessed data.

# Print the number of targets that are 1s, the total number of samples, and the proportion for training, validation, and test.
print(np.sum(train_targets), train_samples_count, np.sum(train_targets) / train_samples_count)
print(np.sum(validation_targets), validation_samples_count, np.sum(validation_targets) / validation_samples_count)
print(np.sum(test_targets), test_samples_count, np.sum(test_targets) / test_samples_count)
```

```
1787.0 3579 0.4993014808605756
235.0 447 0.5257270693512305
215.0 448 0.4799107142857143
```

Save the three dataset in *.npz

In [8]:
```python
# Save the three datasets in *.npz.
# In the next lesson, you will see that it is extremely valuable to name them in such a coherent way!

np.savez('Audiobooks_data_train', inputs=train_inputs, targets=train_targets)
np.savez('Audiobooks_data_validation', inputs=validation_inputs, targets=validation_targets)
np.savez('Audiobooks_data_test', inputs=test_inputs, targets=test_targets)
```

Create the machine learning algorithm

In [9]:
```python
# let's create a temporary variable npz, where we will store each of the three Audiobooks datasets
npz = np.load('Audiobooks_data_train.npz')

# we extract the inputs using the keyword under which we saved them
# to ensure that they are all floats, let's also take care of that
train_inputs = npz['inputs'].astype(np.float64)
# targets must be int because of sparse_categorical_crossentropy (we want to be able to smoothly one-hot encode them)
train_targets = npz['targets'].astype(np.int32)

# we load the validation data in the temporary variable
npz = np.load('Audiobooks_data_validation.npz')
# we can load the inputs and the targets in the same line
validation_inputs, validation_targets = npz['inputs'].astype(np.float64), npz['targets'].astype(np.int32)

# we load the test data in the temporary variable
npz = np.load('Audiobooks_data_test.npz')
# we create 2 variables that will contain the test inputs and the test targets
test_inputs, test_targets = npz['inputs'].astype(np.float64), npz['targets'].astype(np.int32)
```

In [10]:
```python
# Set the input and output sizes
input_size = 10
output_size = 2
# Use same hidden layer size for both hidden layers. Not a necessity.
hidden_layer_size = 50

# define how the model will look like
model = tf.keras.Sequential([
    # tf.keras.layers.Dense is basically implementing: output = activation(dot(input, weight) + bias)
    # it takes several arguments, but the most important ones for us are the hidden_layer_size and the activation function
    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 1st hidden layer
    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 2nd hidden layer
    # the final layer is no different, we just make sure to activate it with softmax
    tf.keras.layers.Dense(output_size, activation='softmax') # output layer
])

### Choose the optimizer and the loss function

# we define the optimizer we'd like to use,
# the loss function,
# and the metrics we are interested in obtaining at each iteration
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

### Training
# That's where we train the model we have built.

# set the batch size
batch_size = 100

# set a maximum number of training epochs
max_epochs = 100

# set an early stopping mechanism
# let's set patience=2, to be a bit tolerant against random validation loss increases
early_stopping = tf.keras.callbacks.EarlyStopping(patience=2)

# fit the model
# note that this time the train, validation and test data are not iterable
model.fit(train_inputs, # train inputs
          train_targets, # train targets
          batch_size=batch_size, # batch size
          epochs=max_epochs, # epochs that we will train for (assuming early stopping doesn't kick in)
          # callbacks are functions called by a task when a task is completed
          # task here is to check if val_loss is increasing
          callbacks=[early_stopping], # early stopping
          validation_data=(validation_inputs, validation_targets), # validation data
          verbose = 2) # making sure we get enough information about the training process
```

```
Epoch 1/100
36/36 - 1s - 39ms/step - accuracy: 0.7706 - loss: 0.5606 - val_accuracy: 0.8591 - val_loss: 0.4403
Epoch 2/100
36/36 - 0s - 4ms/step - accuracy: 0.8815 - loss: 0.3629 - val_accuracy: 0.8591 - val_loss: 0.3698
Epoch 3/100
36/36 - 0s - 3ms/step - accuracy: 0.8894 - loss: 0.3165 - val_accuracy: 0.8725 - val_loss: 0.3425
Epoch 4/100
36/36 - 0s - 3ms/step - accuracy: 0.8921 - loss: 0.2943 - val_accuracy: 0.8770 - val_loss: 0.3201
Epoch 5/100
36/36 - 0s - 4ms/step - accuracy: 0.8961 - loss: 0.2807 - val_accuracy: 0.8859 - val_loss: 0.3062
Epoch 6/100
36/36 - 0s - 4ms/step - accuracy: 0.8986 - loss: 0.2719 - val_accuracy: 0.8881 - val_loss: 0.2961
Epoch 7/100
36/36 - 0s - 3ms/step - accuracy: 0.9016 - loss: 0.2665 - val_accuracy: 0.8881 - val_loss: 0.2905
Epoch 8/100
36/36 - 0s - 4ms/step - accuracy: 0.9042 - loss: 0.2599 - val_accuracy: 0.8926 - val_loss: 0.2832
Epoch 9/100
36/36 - 0s - 4ms/step - accuracy: 0.9044 - loss: 0.2552 - val_accuracy: 0.8971 - val_loss: 0.2793
Epoch 10/100
36/36 - 0s - 3ms/step - accuracy: 0.9064 - loss: 0.2503 - val_accuracy: 0.8971 - val_loss: 0.2762
Epoch 11/100
36/36 - 0s - 3ms/step - accuracy: 0.9053 - loss: 0.2475 - val_accuracy: 0.8949 - val_loss: 0.2744
Epoch 12/100
36/36 - 0s - 3ms/step - accuracy: 0.9061 - loss: 0.2438 - val_accuracy: 0.8971 - val_loss: 0.2693
Epoch 13/100
36/36 - 0s - 3ms/step - accuracy: 0.9078 - loss: 0.2451 - val_accuracy: 0.8949 - val_loss: 0.2691
Epoch 14/100
36/36 - 0s - 3ms/step - accuracy: 0.9089 - loss: 0.2398 - val_accuracy: 0.8993 - val_loss: 0.2638
Epoch 15/100
36/36 - 0s - 3ms/step - accuracy: 0.9106 - loss: 0.2404 - val_accuracy: 0.8926 - val_loss: 0.2806
Epoch 16/100
36/36 - 0s - 3ms/step - accuracy: 0.9092 - loss: 0.2417 - val_accuracy: 0.8971 - val_loss: 0.2638
Epoch 17/100
36/36 - 0s - 3ms/step - accuracy: 0.9111 - loss: 0.2356 - val_accuracy: 0.8971 - val_loss: 0.2676
Epoch 18/100
36/36 - 0s - 4ms/step - accuracy: 0.9111 - loss: 0.2343 - val_accuracy: 0.8971 - val_loss: 0.2626
Epoch 19/100
36/36 - 0s - 4ms/step - accuracy: 0.9128 - loss: 0.2318 - val_accuracy: 0.9016 - val_loss: 0.2593
Epoch 20/100
36/36 - 0s - 3ms/step - accuracy: 0.9120 - loss: 0.2302 - val_accuracy: 0.9038 - val_loss: 0.2643
Epoch 21/100
36/36 - 0s - 4ms/step - accuracy: 0.9123 - loss: 0.2316 - val_accuracy: 0.9016 - val_loss: 0.2596
```

Out[10]:
```
<keras.src.callbacks.history.History at 0x190dff22060>
```

Test The Model

In [11]:
```python
test_loss, test_accuracy = model.evaluate(test_inputs, test_targets)

print('\nTest loss: {0:.2f}. Test accuracy: {1:.2f}%'.format(test_loss, test_accuracy*100.))
```

```
14/14 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8860 - loss: 0.2954

Test loss: 0.28. Test accuracy: 89.73%
```

In [ ]:

In [ ]: