CPSC501

Assignment 2

Main Report

Se Yeon Sim
10102631
November 12, 2022

Main Project Stage 1

Dataset:

- This Heart Failure Prediction Dataset was created by combining different datasets already available independently but not combined before. In this dataset, 5 heart datasets are combined over 11 common features. The five datasets used for its curation are :

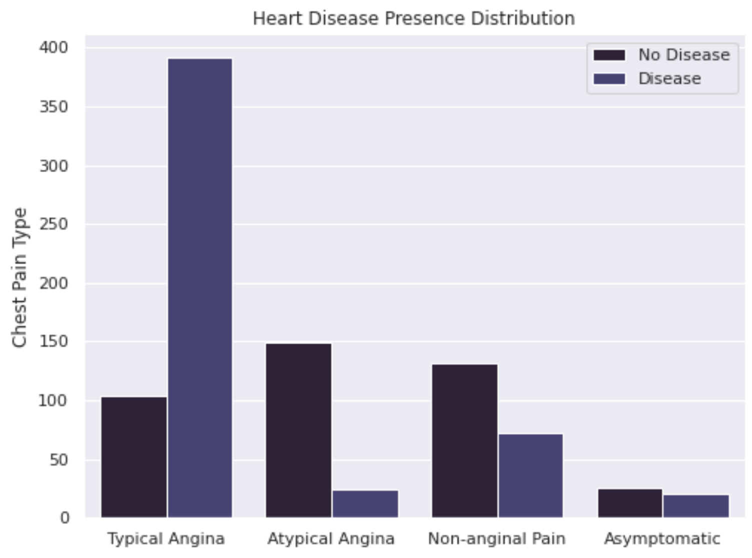
1. Cleveland: 303 observations
2. Hungarian: 294 observations
3. Switzerland: 123 observations
4. Long Beach VA: 200 observations
5. Stalog (Heart) Data Set: 270 observations

- Total: 1190 observations
- Duplicated: 272 observations
- Final dataset: 918 observations
- This Heart Failure Prediction Dataset can be found at https://www.kaggle.com/datasets/fedesoriano/heart-failure-prediction

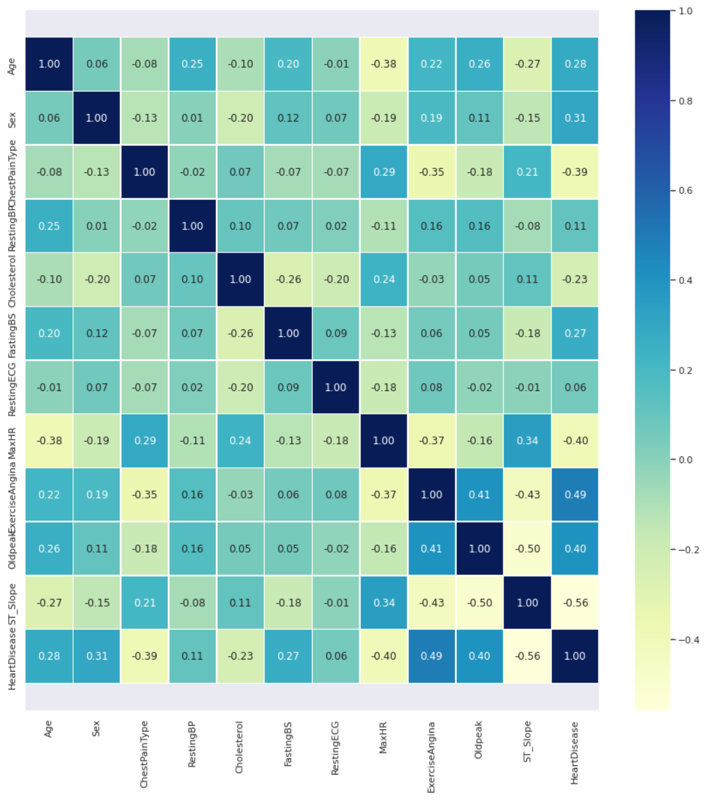
Cleaning Dataset:

- One of the challenges I had in combining dataset was importing dataset directly from Kaggle. To import dataset directly from Kaggle you have to install the Kaggle library and make a directory and copy the kaggle.json into newly made directory. It was a bit complicated but I searched up in google and there were a lot of helpful resources I could find to resolve the issue. The dataset is clear and because It has simple readable column headers and there is no missing data or inconsistent data. And I changed the codes of variables that I classified as categorical and binary.

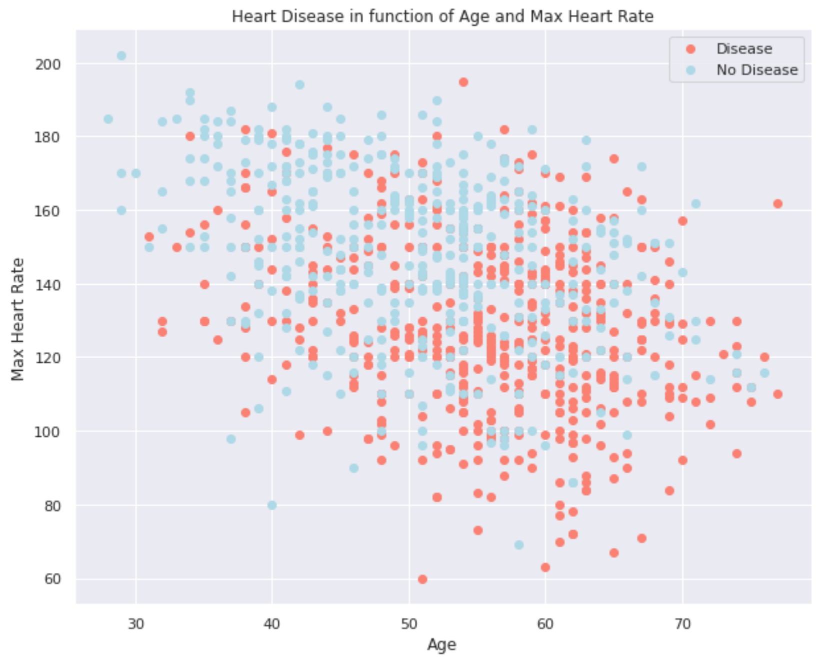
Visualizations:



- This histogram shows heart disease by chest pain types. Axis are all of the “chest pain types” and on the y-axis we have number of heart disease observed and number of no heart disease observed. As you can see patients with “Typical Angina” chest pain type show highest number of heart disease observed and “Atypical Angina” shows highest number of no heart disease observed.



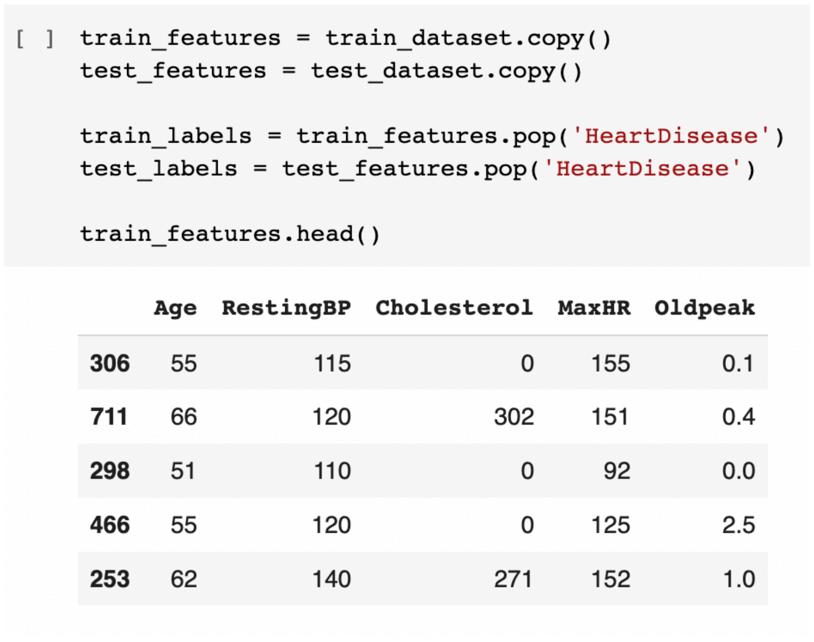
- This heat map shows correlation between each variables related to every variables. As you can see in above heat map. “Age”, “Sex”, “RestingBP”, “FastingBS”, “RestingECG”, “Excercise Angina” and “Oldpeak” are all positively correlated to “HeartDisease”. While “ChestPainType”, “Cholesterol”, “MaxHR”, and “ST_Slope” are all negatively correlated to “HeartDisease”. This makes sense since higher the age result in higher change of heart disease.



- This scatter plot shows Age vs Max Heart Rate for heart disease. Red points all represents patients with heart disease while blue points represents patients with no heart disease. As you can see in the scatterplot above, you can observe more red points between age range of 50-70. You can observe more blue points between age range of 30-50. Also, you can observe downward slope of max heart rate as ages get higher.

Main Project Stage 2

Input columns and output column:

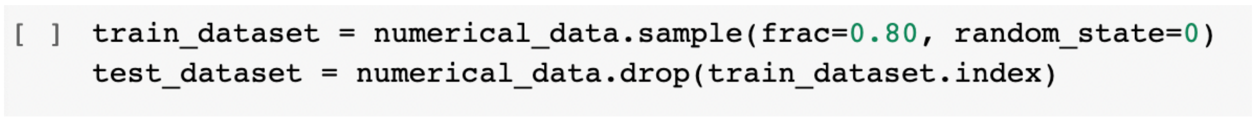


```
[ ] train_features = train_dataset.copy()
    test_features = test_dataset.copy()

    train_labels = train_features.pop('HeartDisease')
    test_labels = test_features.pop('HeartDisease')

    train_features.head()
```

```
     Age  RestingBP  Cholesterol  MaxHR  Oldpeak
306   55        115            0    155      0.1
711   66        120          302    151      0.4
298   51        110            0     92      0.0
466   55        120            0    125      2.5
253   62        140          271    152      1.0
```

For this simple model, I chose 5 numerical columns of “Age”, “RestingBP", “Cholesterol", “MaxHR", and “Oldpeak” as an input columns and I chose “HeartDisease” column as an output column. I chose “HeartDisease” column as output because all other columns seems to related to presence of heart disease. Since all other columns seems important in prediction of heart disease, I decided to use rest of columns as an input columns.

Splitting data into train/test:



```
[ ] train_dataset = numerical_data.sample(frac=0.80, random_state=0)
    test_dataset = numerical_data.drop(train_dataset.index)
```

Train dataset contains 80% of randomly sampled initial dataset and test dataset contains rest of the dataset. Originally I started with 85% of randomly sampled train dataset, however, accuracy was quite low around 60%. I increased and decreased 5% of train dataset each time and retested it. It increased accuracy when I lowered 5% of train dataset. Hence, I decided to go with 80% of train dataset.

Simple model design:

```
import tensorflow as tf
import numpy as np

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(train_features, train_labels, batch_size=128, epochs=30, verbose = 2)
```

I've chosen three layer model as part1 model. The first layer of the model is same as part1 model, simply flattening input into a single dimensional vector. The second layer of the model is a densely connected layer, with 512 outputs. Additionally, the ReLU activation function was chosen because it is simple but works really well with my dataset. The third layer of the model is a densely connected layer with 1 outputs. Additionally, the sigmoid activation function was chosen. Because my output column is binary here, I have a last layer of 1 neuron with sigmoid activation. Sigmoid function guarantee that output unit will always be between 0 and 1.

Hyper-parameters:

For hyper-parameter, I have chosen Adam optimizer because It seems to be quite commonly used, and also works relatively well with my model. For loss function, I have chosen binary cross-entropy because my output is binary. For epochs, I have chosen 30 because model training accuracy keep increasing until 30 train cycles.

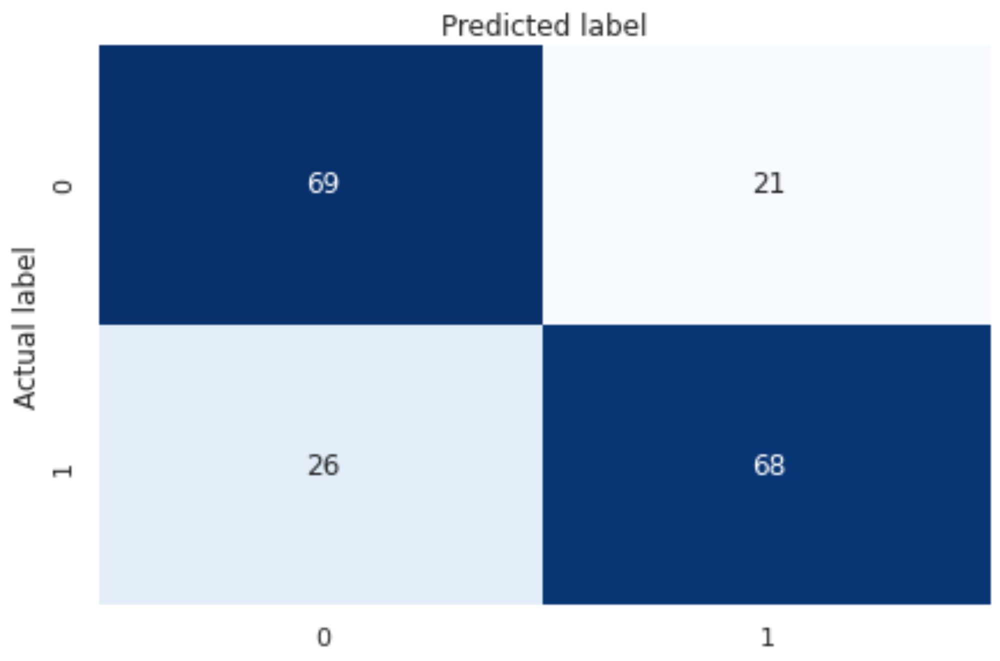
Current accuracy:

My train accuracy is around 73.6% and my test accuracy comes out to be 64.7%. It makes sense because I haven't added categorical and binary columns as an inputs. And model is quite simple with only three layers. Also, I haven't preprocess my input columns and I haven't normalize my input columns. It seems like these are the reasons for low accuracy.

Main Project Stage3

Visualization:

I created confusion matrix, also known as the error matrix, left top panel is ‘True Negative’ which means observation is predicted negative and is actually negative. For my model, I have 69 observations which were ‘True Negative’. The right top panel is ‘False Positive’ which means observation is predicted positive and is actually negative. I have 21 observations which were ‘False Positive’. The left bottom panel is ‘False Negative’ which means observation is predicted negative and is actually positive. I have 26 observations which were ‘False Negative’. The right bottom panel is ‘True Positive’ which means observation is predicted positive and is actually positive. I have 68 observations which were ‘True Positive’.



Additional Changes:

- I added LeakyReLU layer with an alpha value of 0.1. This appears to have a positive effect on model, as it now has a 74.5%
- The LeakyReLu activation function appears to have an advantage over ReLu by giving some difference between wrong predictions.

- I added couple Dropout layers which randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting.
- Dropout is one of the most effective and most commonly used regularization techniques for neural networks. (https://www.tensorflow.org/tutorials/keras/overfit_and_underfit)

```
import tensorflow as tf
import numpy as np

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dropout(rate=0.2),
  tf.keras.layers.LeakyReLU(alpha=0.1),
  tf.keras.layers.Dropout(rate=0.2),
  tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(train_features, train_labels, batch_size=128, epochs=30, verbose = 2)
```

Main Project Stage 4:

Integrating the non-numeric data:

For integrating the non-numeric data I used code from tensorflow tutorial: https://www.tensorflow.org/tutorials/load_data/pandas_dataframe

Since categorical inputs are unordered and they are inappropriate to feed directly to the model; the model would interpret them as being ordered. To use these inputs I needed to encode them, either as one-hot vectors or embedding vectors. For binary features on the other hand do not generally need to be encoded or normalized.

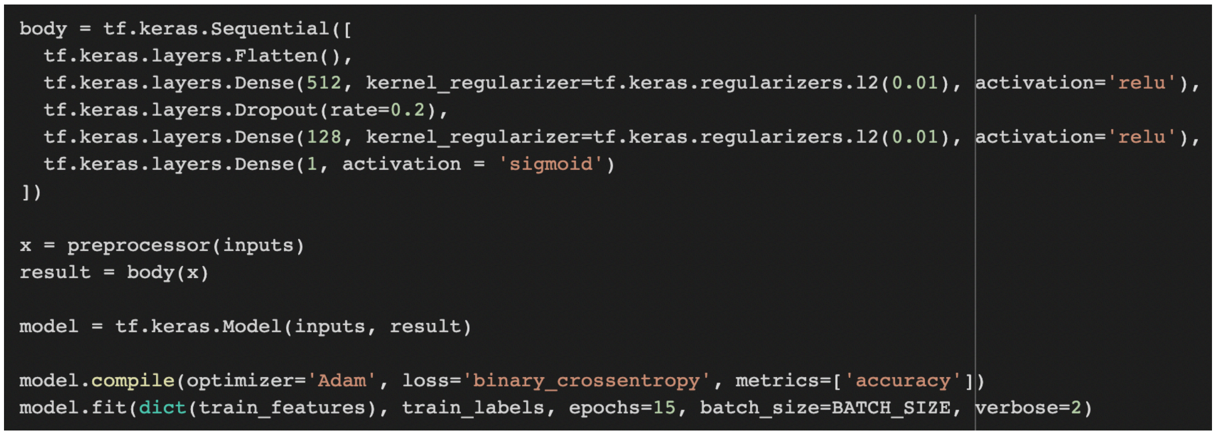
For integrating all of the inputs, I build a preprocessing model that will apply appropriate preprocessing to each input and concatenate the results. Since binary inputs don't need any preprocessing, I just added the vector axis and add them to the list of pre-processed inputs.

For categorical features, I first needed to encode them into either binary vectors or embeddings. Since all these features only contain a small number of categories, I convert the inputs directly to one-hot vectors using the output_mode='one_hot' option. To determine the vocabulary for each input, I create a layer to convert that vocabulary to a one-hot vector

Then, I concatenate all the preprocessed features along the depth axis, so each dictionary-example is converted into a single vector. The vector contains categorical features, numeric features, and categorical one-hot features.

Over-fitting/under-fitting concerns:

For over-fitting/under-fitting concerns I used code from tensorflow tutorial:https://www.tensorflow.org/tutorials/keras/overfit_and_underfit



```
body = tf.keras.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, kernel_regularizer=tf.keras.regularizers.l2(0.01), activation='relu'),
  tf.keras.layers.Dropout(rate=0.2),
  tf.keras.layers.Dense(128, kernel_regularizer=tf.keras.regularizers.l2(0.01), activation='relu'),
  tf.keras.layers.Dense(1, activation = 'sigmoid')
])

x = preprocessor(inputs)
result = body(x)

model = tf.keras.Model(inputs, result)

model.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(dict(train_features), train_labels, epochs=15, batch_size=BATCH_SIZE, verbose=2)
```

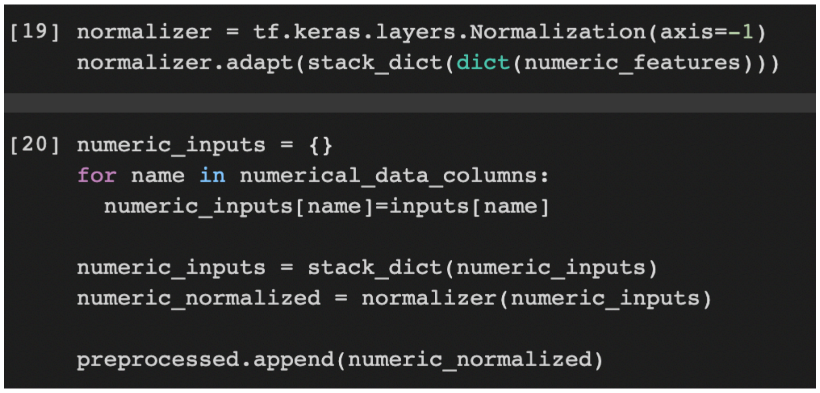
A common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights only to take small values, which makes the distribution of weight values more "regular". This is called "weight regularization", and it is done by adding to the loss function of the network a cost associated with having large weights. I added weight regularization by passing weight regularizer instances to layers as keyword arguments to mitigate overfitting/underfitting.

Also, Dropout is one of the most effective and most commonly used regularization techniques for neural networks. Dropout, applied to a layer, consist of randomly “dropping out” a number of output features of the layer during training. I added dropout layer to my network to prevent overfitting/underfitting.

Model decisions (batch processing, normalization, one-hot-encoding):

For normalization and the one-hot-encoding, I used code from tensorflow tutorial: https://www.tensorflow.org/tutorials/load_data/pandas_dataframe

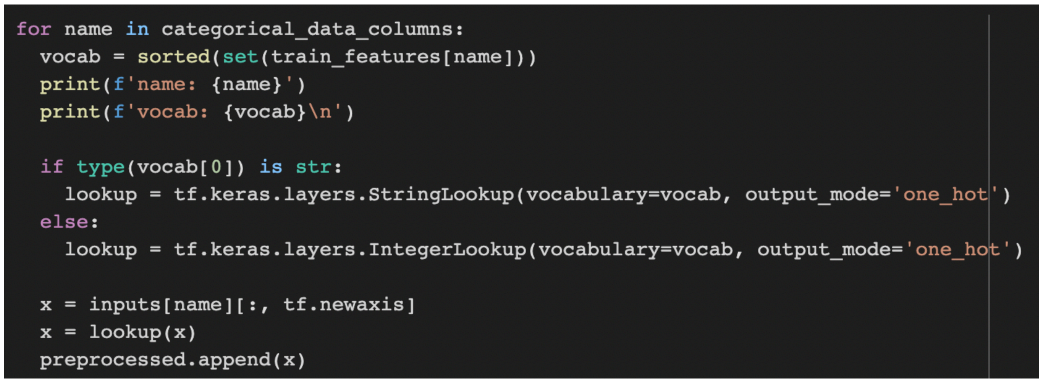
I normalized the numeric input range by using a tf.keras.layers.Normalization layer. To set the layer’s mean and standard-deviation, I called Normalization.adapt method before running it. The code below collects the numeric features from the DataFrame, stacks them together and passes those to the Normalization.adapt method. Then, it stacks the numeric features and runs them through the normalization layer.



```
[19] normalizer = tf.keras.layers.Normalization(axis=-1)
     normalizer.adapt(stack_dict(dict(numeric_features)))
```

```
[20] numeric_inputs = {}
     for name in numerical_data_columns:
       numeric_inputs[name]=inputs[name]

     numeric_inputs = stack_dict(numeric_inputs)
     numeric_normalized = normalizer(numeric_inputs)

     preprocessed.append(numeric_normalized)
```

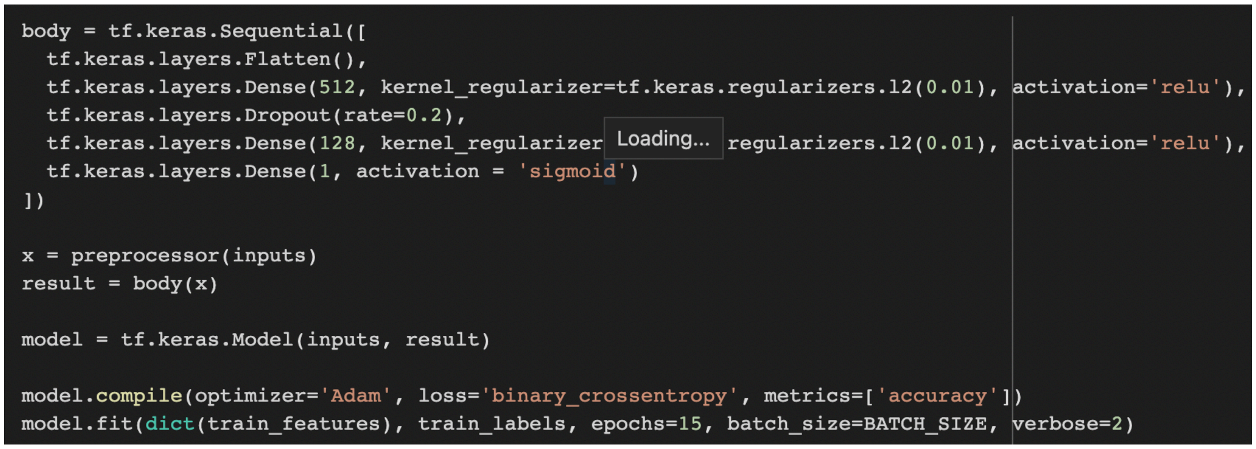
For categorical features, I first needed to encode them into either binary vectors or embeddings. Since all these features only contain a small number of categories, I convert the inputs directly to one-hot vectors using the output_mode=‘one_hot’ option. To determine the vocabulary for each input, I create a layer to convert that vocabulary to a one-hot vector.



```
for name in categorical_data_columns:
  vocab = sorted(set(train_features[name]))
  print(f'name: {name}')
  print(f'vocab: {vocab}\n')

  if type(vocab[0]) is str:
    lookup = tf.keras.layers.StringLookup(vocabulary=vocab, output_mode='one_hot')
  else:
    lookup = tf.keras.layers.IntegerLookup(vocabulary=vocab, output_mode='one_hot')

  x = inputs[name][:, tf.newaxis]
  x = lookup(x)
  preprocessed.append(x)
```

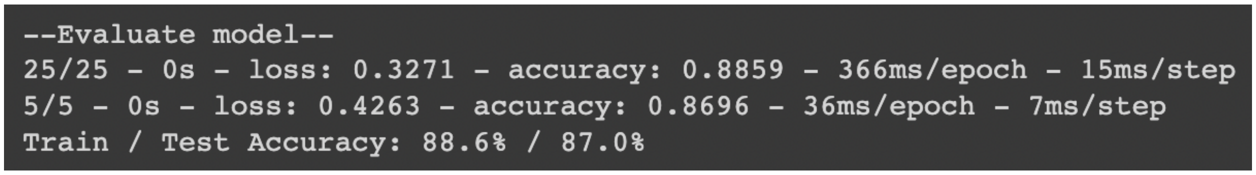
For model, I have 5 layers.

- I added a Dropout layer which randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Dropout is one of the most effective and most commonly used regularization techniques for neural networks. (https://www.tensorflow.org/tutorials/keras/overfit_and_underfit)
- To mitigate overfitting, I put constraints on the complexity of a network by forcing its weights only to take small values, which makes the distribution of weight values more "regular". This is called "weight regularization", and it is done by adding to the loss function of the network a cost associated with having large weights. I added weight regularization by passing weight regularizer instances to layers as keyword arguments to mitigate overfitting/underfitting.
- I have Densely connected layer with ReLu activation function.
- Lastly, I have a Densely connected Layer with sigmoid activation function.



```
body = tf.keras.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, kernel_regularizer=tf.keras.regularizers.l2(0.01), activation='relu'),
  tf.keras.layers.Dropout(rate=0.2),
  tf.keras.layers.Dense(128, kernel_regularizer Loading... regularizers.l2(0.01), activation='relu'),
  tf.keras.layers.Dense(1, activation = 'sigmoid')
])

x = preprocessor(inputs)
result = body(x)

model = tf.keras.Model(inputs, result)

model.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(dict(train_features), train_labels, epochs=15, batch_size=BATCH_SIZE, verbose=2)
```

Result:



```
--Evaluate model--
25/25 - 0s - loss: 0.3271 - accuracy: 0.8859 - 366ms/epoch - 15ms/step
5/5 - 0s - loss: 0.4263 - accuracy: 0.8696 - 36ms/epoch - 7ms/step
Train / Test Accuracy: 88.6% / 87.0%
```

I was able to achieve train accuracy of 88.6% and test accuracy of 87.0%