Sultan Raheem

Instructor: Bharat

Teaching Assistant(s): Sunil Khambaita, Divya Suthraye

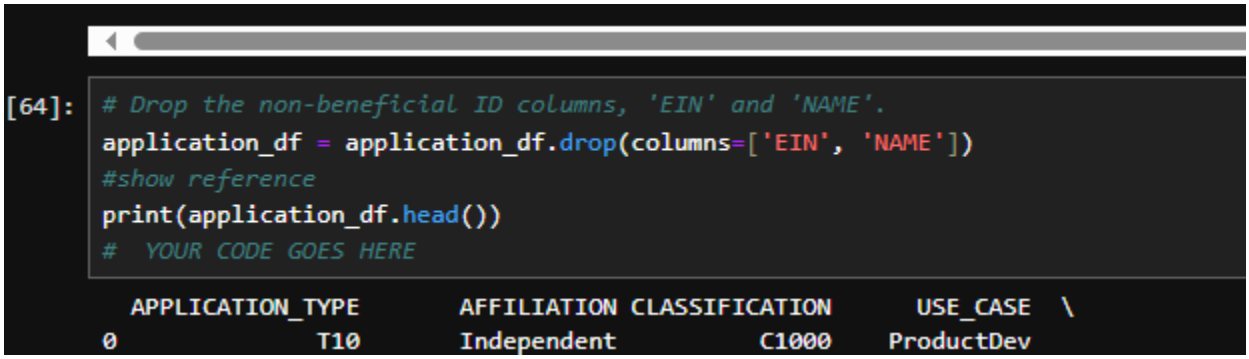
Course: UofT SCS Data Analytics Boot Camp

07-29-2024 - 11:59 pm

In our model, I had CSV data of charity file_name charity_data. The purpose was to look at the given data and create new categories that we would highlight for the Alphabet Soup data. We looked to receive results that would be better than 75% as our threshold.

Variables for the model were through column TARGET.

In the Notebook dropped columns were EIN and NAME.



```
[64]: # Drop the non-beneficial ID columns, 'EIN' and 'NAME'.
      application_df = application_df.drop(columns=['EIN', 'NAME'])
      #show reference
      print(application_df.head())
      #  YOUR CODE GOES HERE

        APPLICATION_TYPE       AFFILIATION CLASSIFICATION      USE_CASE  \
      0              T10       Independent          C1000    ProductDev
```

This is true for both files.

The features concerning the Application_type and Classification columns, which were the cutoff points for each, will be shown below in the notebook.

```
[67]: # Choose a cutoff value and create a list of application types to be replaced
      # use the variable name `application_types_to_replace`
      #  YOUR CODE GOES HERE

      # Replace in dataframe
      for app in application_types_to_replace:
          application_df['APPLICATION_TYPE'] = application_df['APPLICATION_TYPE'].replace(app,"Other")

      # Check to make sure replacement was successful
      application_df['APPLICATION_TYPE'].value_counts()
```

```
[67]: APPLICATION_TYPE
      T3       27037
      T4        1542
      T6        1216
      T5        1173
      T19       1065
      T8         737
      T7         725
      T10        528
      T9         156
      T13         66
      T12         27
      T2          16
      Other       11
      Name: count, dtype: int64
```

```
[68]: # Look at CLASSIFICATION value counts to identify and replace with "Other"
      #  YOUR CODE GOES HERE

      classification_counts = application_df['CLASSIFICATION'].value_counts()
      print(classification_counts)

      threshold = 100
      classifications_to_replace = classification_counts[classification_counts < threshold].index

      application_df['CLASSIFICATION'] = application_df['CLASSIFICATION'].replace(classifications_to_replace, 'Other')

      updated_classification_counts = application_df['CLASSIFICATION'].value_counts()
      print(updated_classification_counts)
```

Compiling, Training, and Evaluating the Model:

In terms of layers, it is different in both files, we can see in the non-optimized version I just have two layers with the output, and it will look like this:

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense_3 (Dense) | (None, 64) | 3,456 |
| dense_4 (Dense) | (None, 32) | 2,080 |
| dense_5 (Dense) | (None, 1) | 33 |

But in the second optimized version it is like this:

Model: "sequential_2"

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense_4 (Dense) | (None, 128) | 6,912 |
| dense_5 (Dense) | (None, 64) | 8,256 |
| dense_6 (Dense) | (None, 32) | 2,080 |
| dropout_1 (Dropout) | (None, 32) | 0 |
| dense_7 (Dense) | (None, 1) | 33 |

Both have the input layer and that was just based on the dimensions

Then there is the First Layer, each layer has the respective neurons written as units within the images, also, in the optimized version, the dropout layers were set to their rate of 0.5. Only the

```
# Second hidden layer
#  YOUR CODE GOES HERE
nn.add(tf.keras.layers.Dense(units=64, activation='relu'))

# 3rd layer
nn.add(tf.keras.layers.Dense(units=32, activation='relu'))

#Layer = dropout (Dropout)
nn.add(tf.keras.layers.Dropout(0.5))
```

```
nn = tf.keras.models.Sequential()

# First hidden layer
#  YOUR CODE GOES HERE
nn.add(tf.keras.layers.Dense(units=64, activation='relu', input_dim=X_train_scaled.shape[1]))


# Second hidden layer
#  YOUR CODE GOES HERE
nn.add(tf.keras.layers.Dense(units=32, activation='relu'))


# Output layer
#  YOUR CODE GOES HERE
nn.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
```

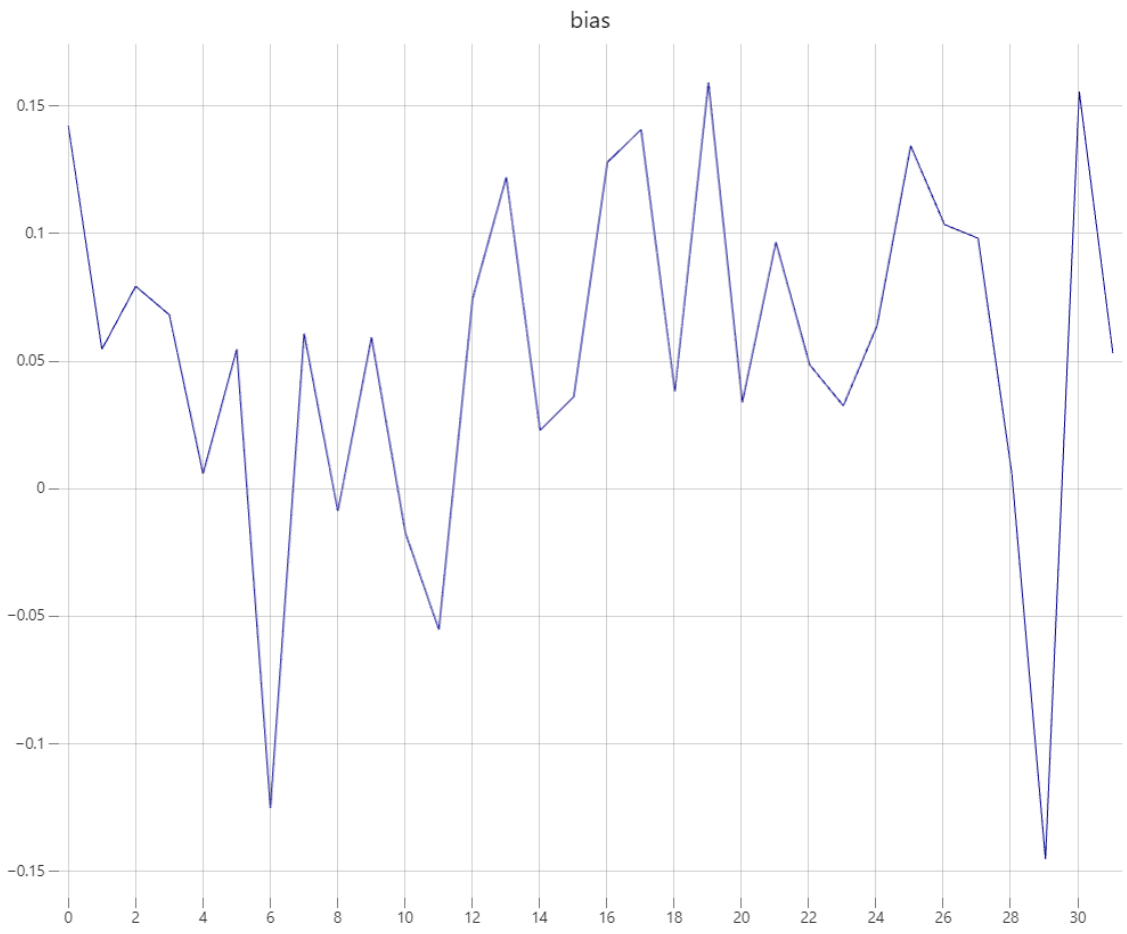
Only the optimized notebook file has the five layers:

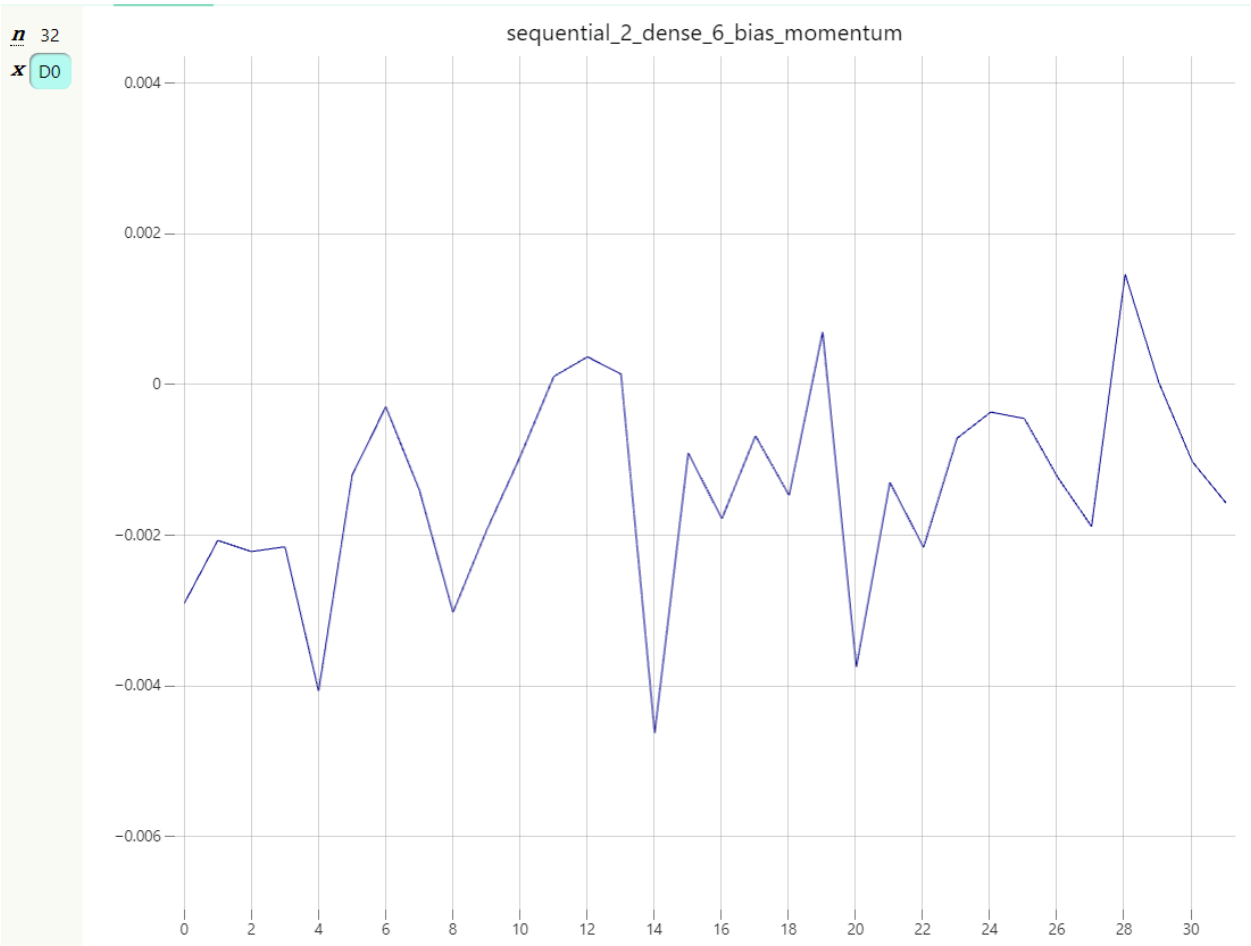
```
# First hidden layer
#  YOUR CODE GOES HERE
nn.add(tf.keras.layers.Dense(units=128, activation='relu', input_dim=X_train_scaled.shape[1]))


# Second hidden layer
#  YOUR CODE GOES HERE
nn.add(tf.keras.layers.Dense(units=64, activation='relu'))

# 3rd layer
nn.add(tf.keras.layers.Dense(units=32, activation='relu'))

#Layer = dropout (Dropout)
nn.add(tf.keras.layers.Dropout(0.5))


# Output layer
#  YOUR CODE GOES HERE
nn.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
```

The only reason we would ever need to increase the layers and the number of neurons is to see the data in our H5 file more broken down and display patterns more accurately.

n 32

x D0






We can summarize that in the model we were analyzing and the data we were given, its accuracy falls above ¾ at 75%<.

Another way we could have done this is GBM, and the reasons I have found for it are that there is no tuning needed within the model. It will be ready to use as is furthermore, it can also be very highly accurate and is a recognized method of modelling.

All the work cited information will be provided in the README of the GitHub submission.