ANÁLISIS Y REPORTE SOBRE EL RENDIMIENTO DEL MODELO SPACESHIP TITANIC

Inteligencia artificial avanzada para la ciencia de datos I (Gpo 101)

Carlos Eduardo Ortega Clement A01707480

10/09/2023

PRIMER MODELO

Para la primera implementación, modifique el dataset y transforme los datos para obtener todas mis dimensiones de forma numérica. Esto con el motivo de poder utilizar todas las variables en el modelo de red neuronal.

El dataset final lo dividí en datos de train, validation y test. Donde en train utilice el 70% de los datos totales y el 30% fue para validation y test, donde se dividió en 50% entre estos.

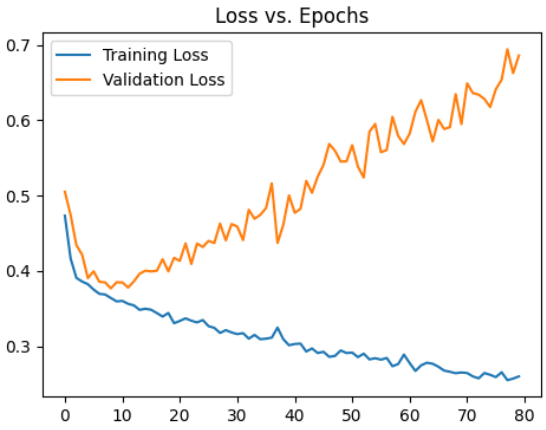
```
#Pasamos nuestros datos a arreglos de numpy
X_data = data_encoded.to_numpy()
Y_data = labels.to_numpy()


# Dividir los datos en conjuntos de entrenamiento, validación y prueba
X_train, X_temp, y_train, y_temp = train_test_split(X_data, Y_data, test_size=0.2, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# Escalar los atributos para un mejor rendimiento
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)


model_nn = tf.keras.Sequential([
    tf.keras.layers.Dense(256, activation='relu', input_shape=(X_train.shape[1],)),
    tf.keras.layers.BatchNormalization(),  # <- Batch normalisation layer
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(256, activation ="relu"),
    tf.keras.layers.BatchNormalization(),  # <- Batch normalisation layer
    tf.keras.layers.Dropout(0.5),    #  <- Dropouts para redicr overfitting
    tf.keras.layers.Dense(128, activation ="relu"),
    tf.keras.layers.BatchNormalization(),  # <- Batch normalisation layer
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(64, activation ="relu"),
    tf.keras.layers.BatchNormalization(),  # <- Batch normalisation layer
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(32, activation ="relu"),
    tf.keras.layers.Dense(1, activation='sigmoid')   # Funcion sigmoid ya que es una salida BOOL
])
# Compilar el modelo
model_nn.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Entrenar el modelo
history = model_nn.fit(X_train, y_train, epochs=80, batch_size=128, verbose=2, validation_data=(X_val, y_val))
```

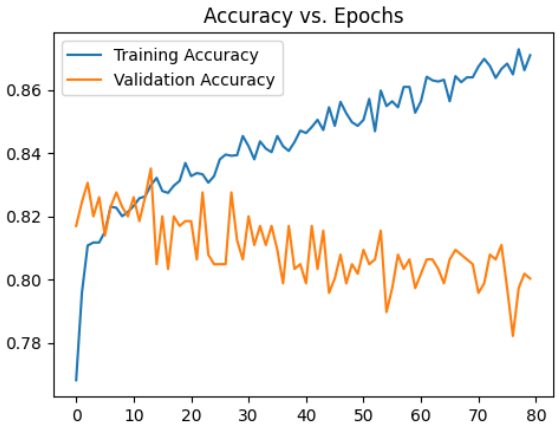
Aquí se puede observar el modelo implementado, donde utilice una red neuronal secuencial. Implementé algunas capas de dropout y batchnormalization para reducir la variabilidad de los

datos y sobre ajsute del modelo ya que antes de implementarlos se podía notar un claro sobre ajuste y mal accuracy en el modelo.

RESULTADO ANTES DEL BATCHNORMALIZATION Y DROPOUT:

```
21/21 [==============================] - 0s 2ms/step
Precisión del modelo de red neuronal en el conjunto de prueba: 0.80
              precision    recall  f1-score   support

           0       0.80      0.79      0.79       321
           1       0.80      0.82      0.81       340

    accuracy                           0.80       661
   macro avg       0.80      0.80      0.80       661
weighted avg       0.80      0.80      0.80       661
```

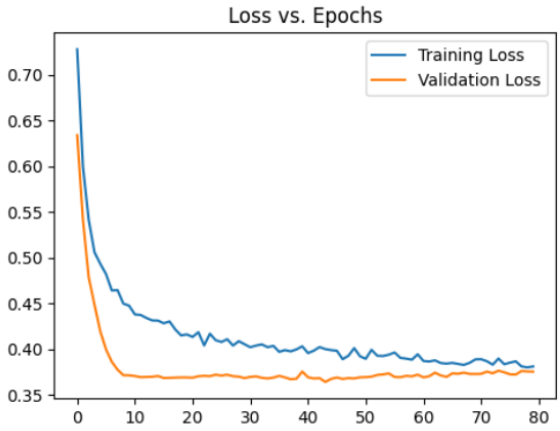


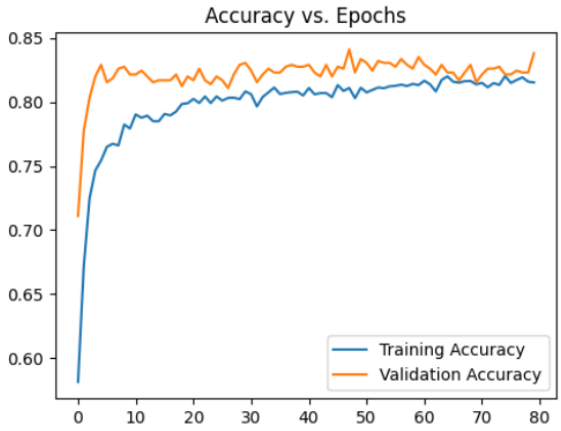


Se puede ver con claridad que a pesar de marcar un 80% de accuracy global, existe un sobreajuste de los datos ya que vemos como la perdida de validation aumente mientras que la de train disminuye, igual con el accuracy, con esto también podemos notar que el sesgo es muy alto. Además, podemos notar mucha fluctuación en el accuracy, tanto en train como en validation, eso quiere decir que existe una varianza muy alta.

RESULTADO CON LAS CAPAS DE DROPOUT Y BATCH NORMALIZATION:

```
21/21 [==============================] - 0s 2ms/step
Precisión del modelo de red neuronal en el conjunto de prueba: 0.82
              precision    recall  f1-score   support

           0       0.82      0.81      0.81       321
           1       0.82      0.83      0.82       340

    accuracy                           0.82       661
   macro avg       0.82      0.82      0.82       661
weighted avg       0.82      0.82      0.82       661
```

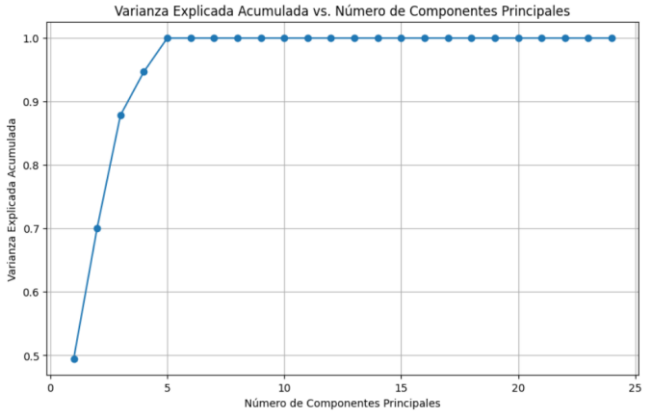




En cambio, después de implementar las capas de dropout y batch normalization, vemos que el sesgo redujo considerablemente y ya no existe un overfitting. Además, alcanzamos un accuracy de 82%. Sin embargo, todavía difieren tanto el validation como el training los y la varianza se redujo, pero sigue siendo medianamente considerable.

Es por eso que realicé algunos ajustes de los datos, primero implementé PCA para reducir la dimensionalidad de los datos y evitar datos que estén altamente correlacionados y puedan afectar el modelo. También ajusté las capas del modelo agregando más neuronas, ajustando el learning rate de forma manual e implementando en algunas capas un kernel regularizer L2, para penalizar los pesos grandes forzados y así reducir la varianza y el sesgo.

Para saber cuántos componentes usar en mi PCA primero obtuve una gráfica de varianza vs componentes y utilicé el método del codo para definir cuantos componentes usar. Al ver la gráfica primero utilicé 5 componentes, pero fui variando entre 4 y 7 componentes y al final obtuve mejor resultado con 7 componentes.



	PC1	PC2	PC3	PC4	PC5	PC6	PC7
0	-599.860671	-242.352924	-11.620874	-262.715829	-116.328396	11.718011	0.525436
1	-443.082868	181.516309	-330.130230	-162.183594	-126.195351	-4.171166	-0.529562
2	4451.537094	3435.200653	-4350.887848	-344.697263	-197.361660	19.801584	-0.542264
3	1486.281732	1857.250071	-2021.337369	-234.086988	204.641715	0.274990	-0.730302
4	-393.801369	149.464276	-381.910858	55.465301	-50.453419	-12.649170	-0.551920
...	...	...	...	...	...	...	...
6601	6215.870436	-1387.034472	-1211.032819	-249.564839	-96.196934	5.723998	0.693171
6602	-599.888901	-242.375727	-11.618813	-262.755401	-116.338842	-9.298331	-0.641655
6603	-604.852925	-211.509543	-35.261397	188.979022	1699.923868	-3.051169	-0.582723
6604	1264.087177	1547.000233	2226.306531	-188.282144	-130.821855	0.404394	-0.850189
6605	3790.982082	-1876.297927	-152.682974	-107.111634	-117.077560	12.327180	-0.677923

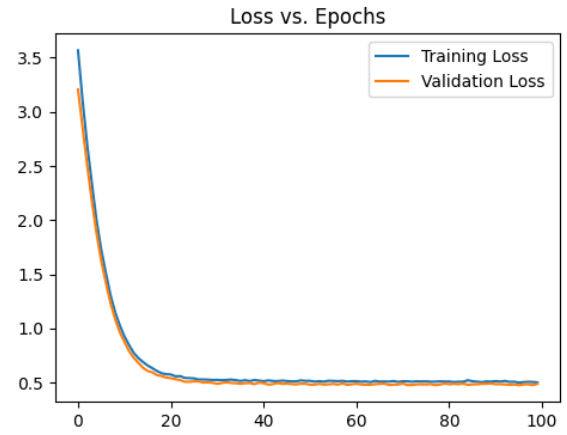
SEGUNDO MODELO O MODELO FINAL

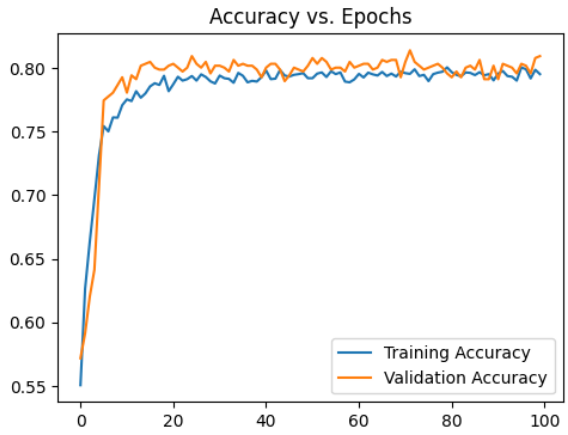
Finalmente, implementé este modelo resultante de modificar los datos para obtener 7 dimensiones con el PCA, los epochs a 100, learning rate a 0.001, batch_size a 128. También modifiqué el número de neuronas y número de capas y finalmente agregar dos capas con el L2 regularizer a 0.008.

```python
# Dividir los datos en conjuntos de entrenamiento, validación y prueba
X_train, X_temp, y_train, y_temp = train_test_split(X_pca, Y_data, test_size=0.2, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# Escalar los atributos para un mejor rendimiento
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)


model_nn = tf.keras.Sequential([
    tf.keras.layers.Dense(512, activation='relu',kernel_regularizer=l2(0.008), input_shape=(X_train.shape[1],)),
    tf.keras.layers.BatchNormalization(),  # <- Batch normalisation layer
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(256, activation ="relu",kernel_regularizer=l2(0.008)),
    tf.keras.layers.Dropout(0.5),    #  <- Dropouts para redicr overfitting
    tf.keras.layers.Dense(128, activation ="relu"),
    tf.keras.layers.BatchNormalization(),  # <- Batch normalisation layer
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(64, activation ="relu"),
    tf.keras.layers.BatchNormalization(),  # <- Batch normalisation layer
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(64, activation ="relu"),
    tf.keras.layers.BatchNormalization(),  # <- Batch normalisation layer
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1, activation='sigmoid')   # Funcion sigmoid ya que es una salida BOOL
])
# Compilar el modelo

custom_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0009)  # ajustamos la taza de aprendizaje
model_nn.compile(optimizer=custom_optimizer, loss='binary_crossentropy', metrics=['accuracy'])

# Entrenar el modelo
history = model_nn.fit(X_train, y_train, epochs=100, batch_size=128, verbose=2, validation_data=(X_val, y_val))
```

RESULTADO MODELO FINAL:

```
21/21 [==============================] - 0s 3ms/step
Precisión del modelo de red neuronal en el conjunto de prueba: 0.80
              precision    recall  f1-score   support

           0       0.82      0.77      0.79       321
           1       0.79      0.84      0.82       340

    accuracy                           0.80       661
   macro avg       0.81      0.80      0.80       661
weighted avg       0.81      0.80      0.80       661
```





Podemos observar que los resultados son mejores que los primeros modelos a pesar de que el accuracy se haya reducido a 0.80 en los datos de test, ya que no existe un overfitting ,el sesgo y la varianza bajaron mucho y el accuracy de validación como de entrenamiento no están alejados uno del otro. Algo que noté es que a pesar de modificar mucho el modelo y los datos, el accuracy

nunca subió más de 83%, por lo que posiblemente una red neuronal no es lo mejor para este problema.

Se podría mejorar el accuracy implementando otro tipo de modelos diferentes como un random forest o una regresión lineal. Sería cuestión de implementarlos y ver cuál da mejor accuracy pero para este proyecto implementamos una red neuronal secuencial y tratamos de mejorarla lo más que se pudo.