Can Artificial Intelligence Help Identify Palliative Care Patients?

Thomas K. Arnold
Senior Data Scientist, Population Health

The assigned task for this study was to do an evaluation of Palliative Care in a mid-sized healthcare environment. Palliative Care is a type of patient care that is used to improve the comfort of patients. Palliative Care is typically prescribed when a patient has an extremely painful condition. Palliative care is often used when patients are nearing "end of life," although this does not fit all palliative care patients. Curative care can continue while a patient is receiving palliative care.

The ultimate goal of the study is to be able to create a matched sample of patients that could be used for comparison purposes. This type of sample matching is called "propensity score matching." The goal would be to create a sample of patients that is statistically similar to the Palliative Care group. For example, a 75 year old female with heart disease and diabetes who had received Palliative Care would be matched to another 75 year old female with heart disease and diabetes who had not received Palliative Care.

One problem with propensity score matching is that the matching process takes longer and longer as the number of matching variables increases. Therefore, a smaller subset of matching variables is needed. The initial set of variables available for matching in this study was just over 700 variables. This number of variables was too large. The smaller subset of variables chosen needed to be fairly accurate replacements for the 700 variables.

Since the goal was dimension reduction, it seemed that principal components analysis (PCA) would be appropriate. PCA was used to create a set of 50 variables that were used in the rest of this study. These were named PCA_1, PCA_2, …, PCA_50.

After obtaining the 50 principal components, a Random Forest model was generated to get the base accuracy and AUC score. From there, the KERAS library was used to find the best fitting machine learning model. The accuracy scores and AUC scores for both the train and test samples were saved and used to guide the machine learning process.

After trying several different KERAS options, it turned out that the models were over-fitting the train sample and under-fitting the test sample. A set of remedies were tried to reduce the overfitting and get the test sample accuracy back up. After a number of attempts, the models were generating over 90% accuracy scores and about 92% AUC scores.

The results indicate that there seems to be a certain amount of art that is involved in creating machine learning models. After many attempts, rewriting the code was becoming tedious, so the process was automated to a large extent. Several tools were used to evaluate the models. Since this was a rather lengthy process, only the highlights will be provided.

Main Objective of the Analysis

The main objective of the analyses was to determine whether a machine learning algorithm could correctly identify which patients had received palliative care. The goals were to find the most accurate model possible. Explainability was not an issue. The main objective was to achieve the highest possible level of predictive validity.

Brief Description of the Data Set

Data on 42,157 patients was collected for this study. This consisted of 4,655 patients assigned to palliative care from 2016 to the present, combined with all other patients who had died from 2016 to the present. The palliative care patients were 11% of the total.

Over 700 different variables had been collected, including age at death, sex, race, ethnicity, cost of care, diagnoses, numbers of visits, procedures administered, chronic conditions, blood pressure, cholesterol, etc. The lab data also included data on variation and trends. Dates of death as well as the days from the start of palliative care were also collected.

Brief Summary of Data Exploration and Feature Engineering

The median age of the patients was 82. Palliative care patients tended to be slightly younger in age. Males and females were about equally represented. Non-white patients tended to die at younger ages than white patients.

One of the first steps in cleaning the data was to remove all columns with NANs and non-numeric data. Columns with fewer than 20 non-zero values were also removed.

Principal components analysis (PCA) was used to reduce the number of variables from around 700 to 50 principal components. This produced an X dataset with 42,157 rows by 50 columns. The y data consisted of a single yes/no binary (0/1) response with 42,157 rows.

The data was initially scaled using the min-max scaler and the data scored from 0-1 was used for the Random Forest model. The data was later normalized for use in the machine learning models using the standard scaler.

A train/test ratio of 75%/25% was used to split the data into training and test samples.

Summary of Deep Learning Model Training

A total of 21 different models were tried. These included one Random Forest base model and 20 Keras machine learning models. The machine learning models fell into two primary groups of models that matched the two main models used in the Keras introduction lab from course 5 in the IBM Machine Learning Professional Certification class (05d_LAB_Keras_Intro.ipynb).

The 05d Keras Intro lab used the PIMA Indians diabetes data, which seemed to be similar in nature to the medical data that was used in this study. There were two primary machine learning models used in the lab. The first was a 2 layer model and the second was a 3 layer model. Several iterations were tried of each Keras model.

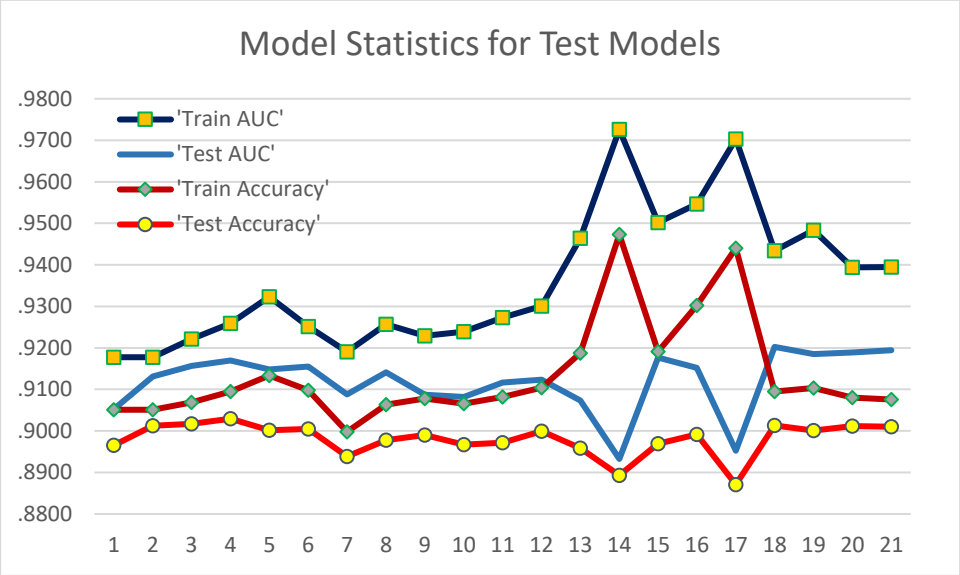
At first, it seemed that the initial Keras model, which only had 2 layers, might be too simple. After trying several 3 layer models, it turned out that the 3 layer models were overfitting the data on the train sample and this was leading to poor accuracy on the test sample. A few more tries on the 2 layer model

showed that it actually does a little better overall in creating the highest accuracy scores, and provides AUC scores that were almost as high as the best 3 layer models. This appeared to be due to the fact that the 2 layer model was less prone to overfitting.

Lessons Learned

There were a few lessons learned. The model accuracies and AUC scores are plotted below. The following table provides basic model descriptions and the train/test accuracy results.

Model Statistics



Model Summaries

Model	Description	Layers	Nodes	Activation	Optimizer	Epochs	Train Accuracy	Test Accuracy	Train AUC	Test AUC
1	Random Forest						.9051	.8966	.9178	.9052
2	2 layer relu, SDG	2	12	relu	SDG	50	.9051	.9012	.9178	.9131
3	2 layer relu, SDG	2	12	relu	SDG	100	.9069	.9017	.9221	.9157
4	2 layer relu, SDG	2	12	relu	SDG	200	.9095	.9029	.9259	.9169
5	2 layer relu, SDG	2	12	relu	SDG	500	.9134	.9002	.9323	.9148
6	2 layer relu, SDG, Early Stopping	2	12	relu	SDG	360	.9098	.9005	.9251	.9155
7	3 layer relu, SDG	3	6/6	relu	SDG	50	.8998	.8938	.9190	.9088
8	3 layer relu, SDG	3	12/12	relu	SDG	50	.9063	.8978	.9257	.9141
9	3 layer LeakyReLU, SDG	3	12/12	LeakyReLU	SDG	50	.9078	.8991	.9229	.9087
10	3 layer relu, SDG	3	18/18	relu	SDG	50	.9066	.8967	.9239	.9082
11	3 layer relu, SDG	3	24/12	relu	SDG	50	.9082	.8972	.9273	.9117
12	3 layer relu, SDG	3	36/12	relu	SDG	50	.9104	.8999	.9300	.9123
13	3 layer, relu, adam	3	12/12	relu	adam	50	.9188	.8958	.9464	.9074
14	3 layer, relu, adam	3	36/12	relu	adam	50	.9473	.8893	.9726	.8933
15	5 layer, relu, adam, .4 dropout	5	36/12	relu	adam	50	.9191	.8970	.9502	.9176
16	5 layer, relu, adam, .4 dropout	5	36/12	relu	adam	100	.9302	.8991	.9547	.9152
17	3 layer, relu, adam, norm	3	36/12	relu	adam	50	.9440	.8871	.9703	.8953
18	5 layer, relu, adam, .5 dropout	5	36/12	relu	adam	50	.9095	.9013	.9434	.9202
19	5 layer, relu, adam, .5 dropout	5	36/12	relu	adam	100	.9104	.9001	.9483	.9185
20	5 layer, relu, adam, .5 dropout	5	36/12	relu	adam	25	.9081	.9011	.9394	.9189
21	5 layer, relu, adam, .5 dropout, ES	5	36/12	relu	adam	27	.9075	.9010	.9394	.9195

Lesson # 1 – Overfitting

One of the more important lessons seems to be that machine learning models are prone to overfitting. There seems to be an inverse relationship between high training accuracy and low test accuracy. This is seen in Models 14 and 17 above. In both cases, the train AUC scores were very high at over 97%, but these models had some of the lowest test accuracies. It appeared that just adding more power to the model was not a good approach. Bigger is not always better.

There were several ways to prevent overfitting. The first 4 were manual and the last 2 were automated.

1. Use plotting to look at outputs.
2. Use fewer nodes.
3. Use fewer layers.
4. Use fewer epochs.
5. Use a DropOut layer.
6. Add an early stopping code to find the ideal number of epochs to train.

The first models tested (Models 1a – 1e) used 2 layers and 12 nodes. These were fairly accurate and did not seem to be as prone to overfitting.

The base code for these models was as follows.

```
## Import Keras objects for Deep Learning
from tensorflow.keras.models  import Sequential
from tensorflow.keras.layers import Input, LeakyReLU, Dense, Flatten, Dropout,
BatchNormalization
from tensorflow.keras.optimizers import Adam, SGD, RMSprop

model_1 = Sequential()
model_1.add(Dense(12,input_shape = (50,),activation = 'sigmoid'))
model_1.add(Dense(1,activation='sigmoid'))

model_1.compile(SGD(lr = .003), "binary_crossentropy", metrics=["accuracy"])
run_hist_1 = model_1.fit(X_train_norm, y_train, validation_data=(X_test_norm, y_test),
epochs=50)
```

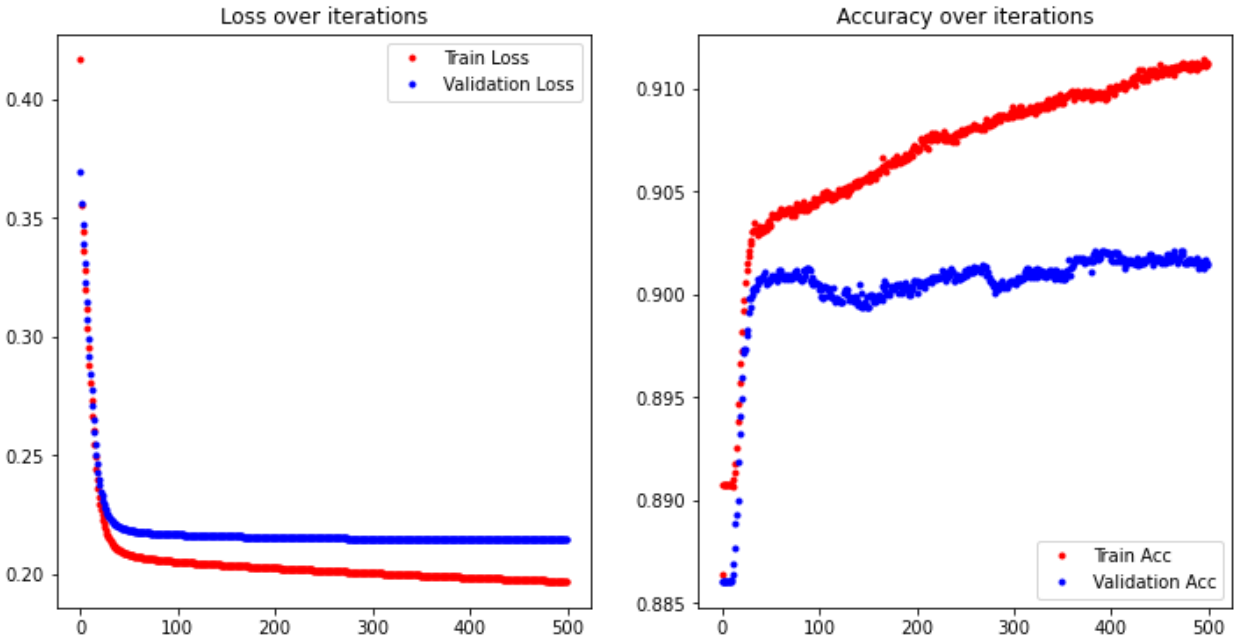
This model improved in accuracy from Model 2 – 4 with the addition of more epochs. Model 4 with 200 epochs was the best performing model. When the code was repeated on Model 5 with 500 epochs, the test accuracy started to drop.

Solution #1 – Plotting and Manual Adjustments

One of the solutions provided in the lab was to use plotting to inspect model output. The results from the code used in the lab for plotting loss and accuracy suggest that Model 5, with 500 epochs, may not be adding to performance. There seems to be some randomness in the output, suggesting that results may be due partly to chance.

With plotting, the manual methods of using fewer nodes, using fewer layers, and using fewer epochs can be tried.

Model 5 Loss and Accuracy on Train and Test Samples



Solution #2 – Add DropOut Layers

An automated solution for overfitting that was found online was to add DropOut layers. This reduces overfitting by removing some of the results between layers. The initial code from the lab for the 3 layer model was as follows.

```
model_2 = Sequential()
model_2.add(Dense(6, input_shape=(50,), activation="relu"))
model_2.add(Dense(6,  activation="relu"))
model_2.add(Dense(1, activation="sigmoid"))

model_2.compile(SGD(lr = .003), "binary_crossentropy", metrics=["accuracy"])
run_hist_2 = model_2.fit(X_train_norm, y_train, validation_data=(X_test_norm, y_test),
epochs=50)
```

After some tweaking, adding “adam” as the optimizer and taking out the stochastic gradient descent, and boosting the model to 36 and 12 nodes respectively, the code looked like this.

```
model.add(Dense(36, input_shape=(50,), activation=”relu”))
model.add(Dense(12,  activation=”relu”))
model.add(Dense(1, activation="sigmoid"))

model.compile(loss="binary_crossentropy", optimizer='adam',  metrics=["accuracy"])
run_hist = model.fit(X_train_norm, y_train, validation_data=(X_test_norm, y_test), epochs=50)
```

At this point, in Model 14, the train sample AUC jumped to over 97%, but the test sample AUC dropped to around 89%. This was less than the previous models and indicated that overfitting was occurring.

One solution to overfitting found on Google was to add DropOut layers. The code is shown below.

```
from tensorflow.keras.layers import Dropout

model.add(Dense(36, input_shape=(50,), activation=”relu”))
model.add(Dropout(0.4))
model.add(Dense(12,  activation=”relu”))
model.add(Dropout(0.4))
model.add(Dense(1, activation="sigmoid"))

model.compile(loss="binary_crossentropy", optimizer='adam',  metrics=["accuracy"])
run_hist = model.fit(X_train_norm, y_train, validation_data=(X_test_norm, y_test), epochs=50)
```

This seemed to help reduce the overfitting as seen in Model 15. A DropOut value of .5 seemed to work even better as seen in the last models (Models 18-21).

Solution #3 – Early Stopping

Another automated solution to prevent overfitting is to add an early stopping callback to the code. This is shown below. The added code is shown in bold.

```
from tensorflow.keras.callbacks import EarlyStopping

es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=20)

model_1e = Sequential()
model_1e.add(Dense(12,input_shape = (50,),activation = 'sigmoid'))
model_1e.add(Dense(1,activation='sigmoid'))
model_1e.compile(SGD(lr = .003), "binary_crossentropy", metrics=["accuracy"])
run_hist_1e = model_1e.fit(X_train_norm, y_train, validation_data=(X_test_norm, y_test),
epochs=500,  callbacks=[es])
```

The results in Model 6 show that with patience = 20, the model stopped after 360 epochs. These results were not as good as the best results as Model 4 with 200 epochs, and so the patience level of 20 might need to be adjusted. This method seems to be very sensitive to random noise. The nice thing about early stopping seems to be that it reduces the chances of overfitting.

Overfitting Summary

As can be seen above, bigger is not always better. There are ways to mitigate the overfitting problem. These solutions seem to be subject to their own sets of problems, and some tweaking is needed to try to get the solutions to work at optimum levels.

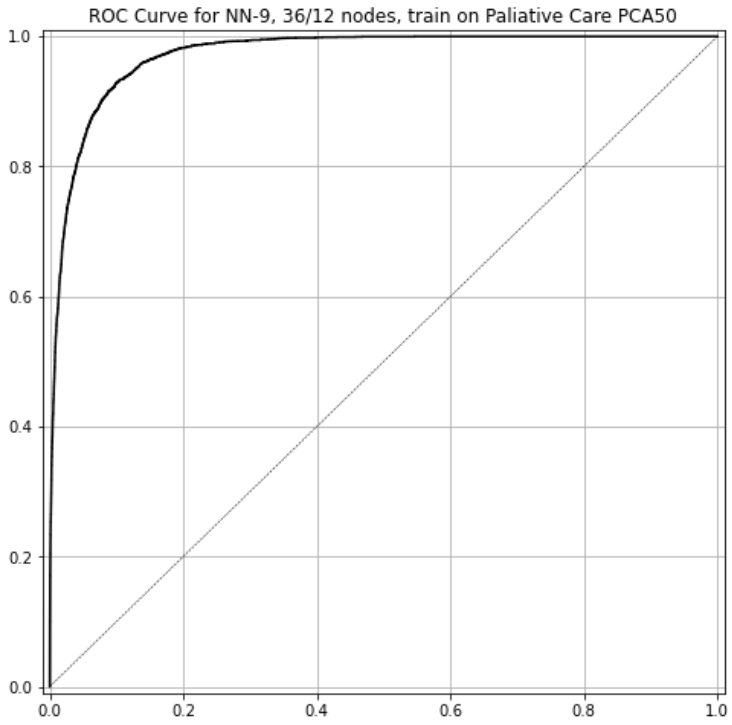
Lesson #2 - Track the Output

Running this many models was confusing and trying to keep the results straight soon became almost impossible. A function was created to save the outputs to a dataframe. This code was as follows.

```
# Create a score dataframe
Scoredf = pd.DataFrame(columns=['Model', 'Accuracy', 'AUC'])

# Create a function to save the score and plot the ROC
def plot_ROC(df, model, X, y, name):
   y_pred_class = model.predict_classes(X)
   y_pred_prob = model.predict(X)

   df = df.append({'Model' : name,
                   'Accuracy' : accuracy_score(y,y_pred_class),
                   'AUC' : roc_auc_score(y,y_pred_prob)},
                   ignore_index=True)

   print('')
   print('accuracy is {:.3f}'.format(accuracy_score(y,y_pred_class)))
   print('roc-auc is {:.3f}'.format(roc_auc_score(y,y_pred_prob)))

   plot_roc(y, y_pred_prob, name)

   return df

# Run the plotting and save routine after the model called “model_name” was run
Scoredf = plot_ROC(Scoredf, model_name, X_test_norm, y_test, 'NN-1, 12 nodes, test')
```

The AUC plot generated for Model 14 with an AUC of over 97% is shown below.

Model 14 AUC Plot



Choosing the "Best" Model

It is difficult to say which is the "best" model at this point. The 2 layer model with 12 nodes performed slightly better on the accuracy scores. The 3 layer model performed slightly better on the AUC scores.

It seemed that there is a lot to learn about building these models. There are numerous methods and options. It seems that a lot of trial and error is required. There are supposed to be automated tuning algorithms built into Keras, but these did not seem to run.

It might be possible to figure out the problems with the Keras tuning algorithms, but it seems that it should not be too difficult to create "home grown" Python coding and automated score capture routines.

I did a little of that already. However, there is only so much that can be accomplished in the short time frame of these classes.

Progress, not perfection.

Summary

In summary, two things seem to be required to get effective machine learning models.

1. Some way to stop overfitting.
2. Some way to automate the process.

It seems that creating powerful models that train nicely on the training set is not very difficult. However, overfitting appears to be a problem. I made some progress on stopping the overfitting.

I tried several methods to avoid overfitting.

1. Use plotting to look at outputs.
2. Use fewer nodes.
3. Use fewer layers.
4. Use fewer epochs.
5. Use a DropOut layer.
6. Add an early stopping code to find the ideal number of epochs to train.

It seems clear that testing multiple models and finding the best solution can become very time consuming. I made some progress on the creation of automation routines. I automated the score capture and also created automated base models that I could tweak by changing parameters.

To progress further, this code would need to be put into a loop so that multiple models could be tested at once.

A good start.

Suggestions

The IBM labs were helpful. I like that there were working code sets to start from. I was able to modify these and tweak them because they worked “out of the box.”

Some of the machine learning code on the Internet seems to be flaky. For example, I could not get the Keras Tuner to run. That would have been useful.

I was not sure if the GridSearchCV would work with automating the search for optimal hyper parameters. It is hard to tell without trying to see whether it would work with Keras.

So much to learn, so little time.

Really, the primary solution to these problems would seem to be more practice.