TabNet — Deep Neural Network for Structured, Tabular Data

Ryan Burke Jul 12, 2021 · 7 min read ★

Photo by Ricardo Gomez Angel on Unsplash

In this post, I will walk you through an example using Google's TabNet for a classification problem.

Despite the recent explosion of Deep Neural Nets (DNNs) for image, audio, and text, it seems like tasks using good ol' structured tabular data have been somewhat ignored.

While it's true that much of the data today is unstructured (approximately 80%), it's important to put in perspective that the *measly* 20% of data bound to rows and columns still represents a MASSIVE volume. In fact, in 2020, IBM estimated that the world's entire data collection to be 35 zettabytes (or 35 billion terabytes).

This leaves **7,000,000,000,000,000,000,000** bytes of structured data needing some Deep Neural Attention!

In all fairness, as pointed out in TabNet's original paper, this is due to the fact the the current ensemble decision tree (DT) variants (XGBoost, LightGBM, CatBoost, etc.,) have some advantages over DNNs for tabular data.

All that vanished, however, since TabNet's release where it has outperformed the DT-based models in multiple benchmark datasets.

Framingham Heart Study

Today, I'm going to go through an example of how to use TabNet for a classification task. The dataset contains results from the Framingham Heart Study, which is a study that began in 1948 and has provided (and is still providing) significant insights into risk factors for cardiovascular disease. For those interested in learning more about this study, please checkout this link.

I you're interested in learning more about TabNet's architecture, I encourage you to look over the original paper I linked above. Additional resources include this repo where you can see the original TabNet code.

Finally, before we dive in, you can follow along using my notebook found at this repo.

The Data

The data used for this analysis consisted of 16 variables, including the target variable ANYCHD. The descriptions of each can be found below.

Search this file…

Variable	Description
SEX	Gender (1 = male / 2 = female)
TOTCHOL	Serum total cholesterol (mg/dL)
AGE	Age at exam (years)
SYSBP	Systolic blood pressure (mmHg)
DIABP	Diastolic blood pressure (mmHg)
CURSMOKE	Current smoker (0 = no / 1 = yes)
CIGPDAY	Cigarettes per day
BMI	Body mass index (kg/m^2)
DIABETES	(0 = no / 1 = yes)
BPMEDS	Blood pressure meds (0 = no / 1 = yes)
HEARTRTE	Heart rate (beats per minute)
GLUCOSE	Serum Glucose (mg/dL)
educ	Education (1 = no high school / 2 = high school / 3 = some college / 4 = college degree
HDLC	HDL cholesterol (mg/dL)
LDLC	LDL cholesterol (mg/dL)
ANYCHD	Incidence of coronary heart disease (0 = no / 1 = yes)

framingham_desc.csv hosted with ❤ by GitHub view raw

Table 1 — Description of the variables found in the data set.

Here's what our dataframe looks like.

Search this file…

SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY	BMI	DIABETES	BPMEDS	HEARTRTE	GLUCOSE
1	209.0	52	121.0	66.0	0	0.0		0	0.0	69.0	92.0
2	237.0	58	108.0	66.0	0	0.0	28.5	0	0.0	80.0	71.0
2		58	155.0	90.0	1	30.0	24.61	0	0.0	74.0	
2	220.0	55	180.0	106.0	0	0.0	31.17	1	1.0	86.0	81.0
2	320.0	57	110.0	46.0	1	30.0	22.02	0	0.0	75.0	87.0
1	280.0	64	168.0	100.0	0	0.0	25.72	0	0.0	92.0	82.0
1	211.0	55	173.0	123.0	0	0.0	29.11	0	1.0	75.0	85.0
2	291.0	62	120.0	70.0	0	0.0	21.98	0		62.0	83.0
2	159.0	53	124.0	78.0	0	0.0	26.62	0	0.0	68.0	135.0
2	264.0	51	141.0	81.0	1	15.0	24.77	0	0.0	85.0	97.0

framingham.csv hosted with ❤ by GitHub view raw

Table 2 — A view of the data in tabular format

Investigating missing values

Next, I wanted to see how much of the data was missing. This is easy to do using df.isnull().sum(), which will tell us how much data is missing per variable. Another way is to use a package missingno which allows us to visualize the relationship between the missing data very quickly.

In figure 1, a matrix representation of the missing values (white) by variable. This is organized vertically by row, which allows us to see if there are any relationships between the missing values. For example, missing values for HDLC and LDLC are identical, suggesting that these values weren't collected for a portion of the patients in this dataset.

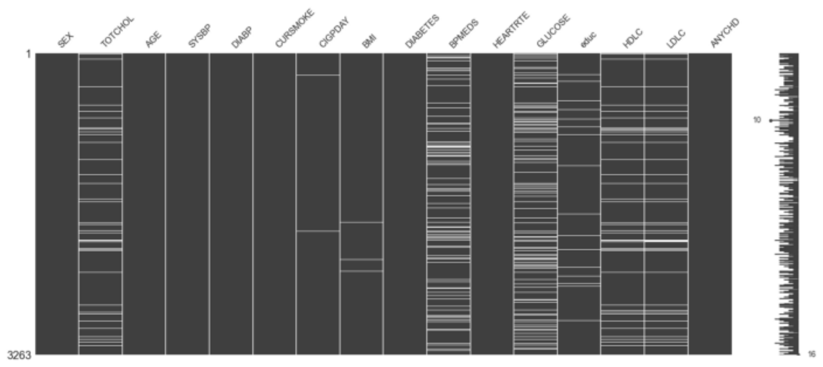

Figure 1 — A matrix of the missing data by variable. Image by author

We can also get a heatmap for a different way of looking at the relationship between missing values as in figure 2. Here we have an easier time seeing the relationship between both HDLC and LDLC with TOTCHOL. The value <1 means that it is slightly less than 1. Since al 3 of these variables are measure of cholesterol, it suggests that cholesterol data was not collected for certain patients in the dataset.

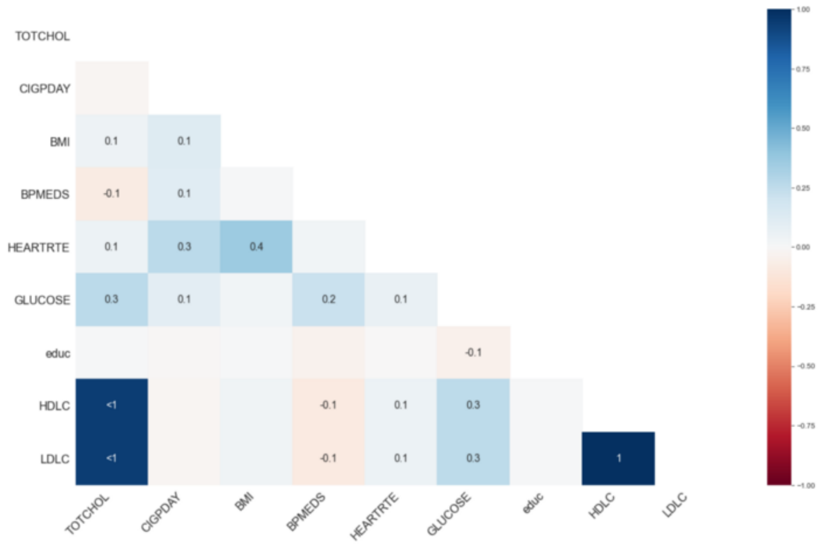

Figure 2 — A heatmap demonstrating the relationship between missing values. Image by author

Imputing missing values

Now that we have gathered information about our missing values, we need to decide what to do about them. There are many options depending on your data, and you can read more about the various imputation algorithms available on sklearn's webpage.

I opted for the KNN imputer, which you can implement using the following code. To summarize, in the first block I simply divided the data into features and target.

The second block transforms the features using the KNN imputer. As you can see from the print statements, there were originally 1812 missing values which were imputed.

```
# separate our target and features variables
target = np.array(df2['ANYCHD'])
# Remove the target from the features
# axis 1 refers to the columns
features= df2.drop('ANYCHD', axis = 1)
# Saving feature names for later use
feature_list = list(features.columns)
# Convert to numpy array
features = np.array(features)

from numpy import isnan
from sklearn.impute import KNNImputer
# print total missing
print('Missing: %d' % sum(isnan(features).flatten()))
# define imputer
imputer = KNNImputer(missing_values=np.nan)
# fit on the dataset
imputer.fit(features)
# transform the dataset
features_trans = imputer.transform(features)
# print total missing
print('Missing: %d' % sum(isnan(features_trans).flatten()))

Missing: 1812
Missing: 0
```

framingham_knn_imputer.png hosted with ❤ by GitHub view raw

Figure 3 — Using the KNN Imputer to deal with missing values in the dataset.

The final step is to split our data. Using the code below, I initially split the data into 70% for the training set and 30% for the validations set. Then I split the validation set in two equal parts for the validation and test sets. The print statements provide us with information about the shape of the splits.

```
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(features_trans, target, test_size=0.30, random_

x_val, x_test, y_val, y_test = train_test_split(x_val, y_val, test_size=0.50, random_state=8)

print("X train shape: ", x_train.shape)
print("X validation shape: ", x_val.shape)
print("X test shape: ", x_test.shape)
print("Y train shape: ", y_train.shape)
print("Y validation shape: ", y_val.shape)
print("Y test shape: ", y_test.shape)

X train shape:  (2284, 20)
X validation shape:  (489, 20)
X test shape:  (490, 20)
Y train shape:  (2284,)
Y validation shape:  (489,)
Y test shape:  (490,)
```

train_val_test_framingham.png hosted with ❤ by GitHub view raw

Figure 4 — Splitting the data into train, validation and test sets

TabNet

You can be ready to run TabNet in a few simple lines, as shown below. This is a pytorch implementation of TabNet, so you'll have to import (or install if you haven't yet) torch, pythorch_tabnet, and the model you wish to use (binary classifier, multi-classifier, regressor).

You will also need some kind of metric to evaluate your model. Here's a list of those available from sklearn. I also included a label enconder in the event that your data is slightly different to mine. My categorical variables are all binary integers, but if you had categories stored as strings you would use this (or an alternative such as one-hot encoding) first.

```
import pytorch_tabnet
from pytorch_tabnet.tab_model import TabNetClassifier
import torch

from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import roc_auc_score, accuracy_score


```

Figure 5 — Importing the necessary libraries

Next, we have to define our model, which you can see in the first block of code below. On the first line we define our optimizer, Adam. The next few lines are scheduling a stepwise decay in our learning rate. Let's unpack what it says:

- The learning rate is initially set to, lr = 0.020
- After 10 epochs, we will apply a decay rate of 0.9
- The result is simply the product of our learning rate and decay rate 0.02*0.9, meaning at epoch 10 it will reduce to 0.018

In the next block of code, we fit the model to our data. Basically it says the train and validation sets will be evaluated with *auc* (area under the curve) and *accuracy* as metrics for a total of 1,000 iterations (*epochs*).

The *patience* parameter states that if an improvement in metrics is not observed after 50 consecutive epochs, the model will stop running and the best weights from the best epoch will be loaded.

The *batch size* of 256 was selected based on recommendations from TabNet's paper, where they suggest a batch size of up to 10% of the total data. They also recommend that the *virtual batch size* is smaller than the batch size and can be evenly divided into the batch size.

The number of *workers* was left at zero, which means that batch sizes will be loaded as needed. From what I read, increasing this number is a very memory-hungry process.

Weights can be 0 (no sampling) or 1 (automated sampling). Lastly, *drop_last* refers to to dropping the last batch if not complete during training.

It’s important to note that many of these are default parameters. You can check out the full list of parameters here.

```
# define the model
clf1_nopreproc = TabNetClassifier(optimizer_fn=torch.optim.Adam,
                       optimizer_params=dict(lr=2e-2),
                       scheduler_params={"step_size":10, # how to use learning rate scheduler
                                         "gamma":0.9},
                       scheduler_fn=torch.optim.lr_scheduler.StepLR,
                       mask_type='entmax' # "sparsemax"
                      )

# fit the model
clf1_nopreproc.fit(
    x_train,y_train,
    eval_set=[(x_train, y_train), (x_val, y_val)],
    eval_name=['train', 'valid'],
    eval_metric=['auc','accuracy'],
    max_epochs=1000 , patience=50,
    batch_size=256, virtual_batch_size=128,
    num_workers=0,
    weights=1,
    drop_last=False
)


```

Figure 6 — Defining and fitting our TabNet classifier

The results of this analysis can be seen in figure 8, and they can be reproduced using the code below. The first three blocks of code plot the loss score, accuracy (for the train and validation sets), and feature importance (for the test set).

The final blocks simply compute the best accuracy achieved for the validation and test sets, which were 68% and 63%, respectively.

```
# plot losses
plt.plot(clf1_nopreproc.history['loss'])

# plot accuracy
plt.plot(clf1_nopreproc.history['train_accuracy'])
plt.plot(clf1_nopreproc.history['valid_accuracy'])

# find and plot feature importance
y_pred = clf1_nopreproc.predict(x_test)
clf1_nopreproc.feature_importances_
feat_importances = pd.Series(clf1_nopreproc.feature_importances_, index=feat.columns)
feat_importances.nlargest(20).plot(kind='barh')


# determine best accuracy for test set
preds = clf1_nopreproc.predict(x_test)
test_acc = accuracy_score(preds, y_test)

# determine best accuracy for validation set
preds_valid = clf1_nopreproc.predict(x_val)
valid_acc = accuracy_score(preds_valid, y_val)

print(f"BEST ACCURACY SCORE ON VALIDATION SET : {valid_acc}")
print(f"BEST ACCURACY SCORE ON TEST SET : {test_acc}")

>>>BEST ACCURACY SCORE ON VALIDATION SET : 0.6809815950920245
>>>BEST ACCURACY SCORE ON TEST SET : 0.6265306122448979

```

Figure 7 — Plotting the loss score, accuracy, and feature importance

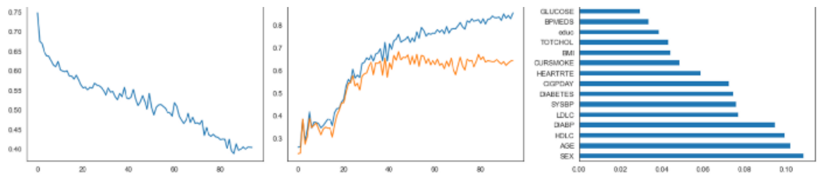

Figure 8 — (left) loss score; (middle) accuracy for training (blue) and validation (orange) sets; (right) relative feature importance

Unsupervised pretraining

TabNet can also be pretrained as an unsupervised model. Pretraining involves deliberately masking certain cells and learning relationships between these cells and adjacent columns by predicting the masked values. The weights learned can then be saved and used for a supervised task.

Let’s see how using unsupervised pretraining can influence our models accuracy!

Although similar, the code has some differences, so I have included it below. Specifically, you have to import the TabNetPretrainer. You can see in the first block of code, the TabNetClassifier is replaced by the TabNetPretrainer.

When you fit the model, note the last line *pretraining_ratio*, which is the percentage of features that are masked during pretraining. A value of 0.8 indicates 80% of features are masked.

The next block of code refers to the reconstructed features generated from TabNet’s encoded representations. These are saved and can then used in a separate supervised task.

```
from pytorch_tabnet.pretraining import TabNetPretrainer

# TabNetPretrainer
unsupervised_model_no_preproc = TabNetPretrainer(
    optimizer_fn=torch.optim.Adam,
    optimizer_params=dict(lr=2e-2),
    mask_type='entmax', # "sparsemax",
    )

# fit the model
unsupervised_model_no_preproc.fit(
    x_train,
    eval_set=[x_val],
    max_epochs=1000 , patience=50,
    batch_size=256, virtual_batch_size=128,
    num_workers=0,
    drop_last=False,
    pretraining_ratio=0.8,

)

# Make reconstruction from a dataset
reconstructed_X, embedded_X = unsupervised_model_no_preproc.predict(x_val)
assert(reconstructed_X.shape==embedded_X.shape)

unsupervised_model_no_preproc.save_model('./test_pretrain2')
loaded_pretrain = TabNetPretrainer()
loaded_pretrain.load_model('./test_pretrain2.zip')


```

tabnet_unsup.png hosted with ❤ by GitHub view raw

Figure 9 — Unsupervised representation learning with TabNet

When pretraining was used for this dataset, the results were 76% and 71% accuracy for the validation and test sets, respectively. This is a significant improvement! Below, you can see the loss scores, accuracy for training (blue) and validation (orange) sets, and the feature importance determined for the test set.

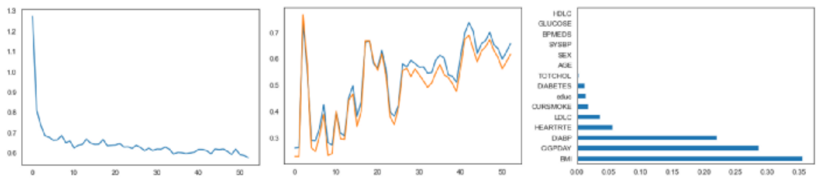

Figure 10 — (left) loss score; (middle) accuracy for training (blue) and validation (orange) sets; (right) relative feature importance. Image by author.

Summary

In this post, we walked through an example of how to implement TabNet for a classification task. We found that unsupervised pretraining with TabNet significantly improved the model's accuracy.

In figure 11 below, I plotted the feature importance for the supervised (left) and unsupervised (right) models. It's interesting that unsupervised pretraining was able to improve the model's accuracy while reducing the number of features.

This makes sense when we think about the relationship of the features to one another. Pretraining with TabNet learns, for example, that blood pressure meds (BPMEDS), systolic blood pressure (SYSBP) and diastolic blood pressure (DIABP) are related. Thus, unsupervised representation learning acts as a superior encoder model for a supervised learning task, with cleaner and more interpretable results.

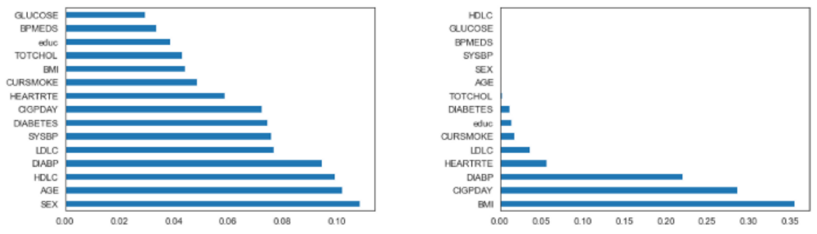

Figure 11 — Feature importance values for the supervised (left) and unsupervised (right) TabNet models. Image by author.

I hope you enjoyed this post! Give it a try and let me know how it worked for you.

Deep Learning   Structured Data   Neural Networks   Classification   Unsupervised Learning