Summer Project Journal

SID LAMSAL

Specifics

Dataset:

- Telco Customer Churn
- predict churn

Applications used: R-Studio, Jupyter Notebook

Base Log Model

Fit

```
[1] "best accuracy:  0.807111111111111"
[1] "best threshold:  0.54"
> evaluateModel(LogModel$fitted.values, bestThresh, train)
   predictionClass
      0    1
  0 3782  333
  1  752  758
[1] "accuracy: 0.807111111111111"
[1] "TP Rate:  0.501986754966887"
```

Predict

```
[1] "best accuracy:  0.815209665955935"
[1] "best threshold:  0.14"
> evaluateModel(predictions, bestThresh, test)
   predictionClass
      0   1
  0 951  97
  1 163 196
[1] "accuracy: 0.815209665955935"
[1] "TP Rate:  0.545961002785515"
```

Fairly similar fit and predictions. Model is significantly better at predicting “No”. “Yes” is 50/50 (i.e. Very bad)

R Neural Network

Package: neuralnet

Function: neuralnet()

To Edit source code: trace(name_of_function, edit = T)

Data Prep For NN

- Had to change all categorical variables to dummy variables with
  - Class 1 = in that category
  - Class 0 = not in that category
- Scaled vars
- Dummy variable for n-1 categories

```
vars <- names(clean[,-20])
remv <- c()

for (var in vars){
  if (is.factor(clean[,var])){
    remv <- c(remv, var)
    cats <- unique(clean[,var])
    for (cat in cats[-1]){
      clean[newVarName(var, cat)] <- ifelse(clean[,var]==cat, 1, 0)
    }
  }
}

clean <- clean[,!(names(clean) %in% remv)]
clean$tenure <- scale(clean$tenure)
clean$MonthlyCharges <- scale(clean$MonthlyCharges)
clean$TotalCharges <- scale(clean$TotalCharges)
```

Base NN

Fit

```
[1] "best accuracy:  0.822933333333333"
[1] "best threshold:  0.49"
> evaluateModel(predictions[,2], bestThresh, train)
   predictionClass
       0    1
  0 3774  364
  1  632  855
[1] "accuracy: 0.822933333333333"
[1] "TP Rate:  0.574983187626093"
> plot(model)
```

Predict

```
[1] "best accuracy:  0.799573560767591"
[1] "best threshold:  0.49"
> evaluateModel(predictions[,2], bestThresh, test)
   predictionClass
      0   1
  0 918 107
  1 175 207
[1] "accuracy: 0.799573560767591"
[1] "TP Rate:  0.541884816753927"
```

2 layers with 4 and 2 neurons and threshold =0.1 (~28sec to train) (not the ideal model but still can measure improvements probably)

Switched to Python

R -> Python

```
#Drop ID
clean = churn.drop("customerID", axis=1)

#Remove missing values
clean.replace(' ', np.nan, inplace=True)
print("Before:", len(clean))
clean = clean.dropna(how= 'any', axis=0)
print("After:", len(clean))

#set data types
clean["TotalCharges"] = clean["TotalCharges"].astype(float)
clean["SeniorCitizen"] = clean["SeniorCitizen"].astype(object)
```

```
Before: 7043
After: 7032
```

```
dummies = pd.get_dummies(clean, drop_first= True)
clean = dummies
clean.dtypes
```

```
yprob = model.predict_proba(xtest)
threshold = 0.5
ypred = (yprob[:, 1] > threshold).astype(int)
ypred
```

```
array([0, 0, 1, ..., 0, 0, 0])
```

```
accuracy = accuracy_score(ytest, ypred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.7853589196872779
```

```
def evaluate(acc, pred):
    cm = confusion_matrix(acc, pred)
    print("Confusion Matrix:")
    print(cm)
    truePositive = cm[1, 1]
    trueNegative = cm[0, 0]
    falsePositive = cm[0, 1]
    falseNegative = cm[1, 0]
    print("\nTrue Positive:", truePositive)
    print("True Negative:", trueNegative)
    print("False Positive:", falsePositive)
    print("False Negative:", falseNegative)
```

```
evaluate(ytest,ypred)
```

```
Confusion Matrix:
[[917 116]
 [186 188]]

True Positive: 188
True Negative: 917
False Positive: 116
False Negative: 186
```

Possible Ways to Use Log Results

Oversample

1. Since “No” is significantly easier to predict, we could over/under sample.

2. Using the new variables, we can get all the missed units and oversample them.

3. We could do a hybrid where we oversample only the class “Yes” units that the model missed.

Churn	prob	class
No	0.101499514	Yes
No	0.564491809	Yes
No	0.072016400	No
No	0.267851655	Yes
No	0.028452609	No
No	0.193505177	Yes
No	0.007850303	No

```
###Simple oversample###

class1 = probdf[probdf["Churn_Yes"] == 1]
distribution = dist(probdf)

oversample = class1.sample(n=(distribution[1]-distribution[0]), replace = True)

overDf = pd.concat([probdf, oversample], ignore_index=True)

dist(overDf)
```

```
Churn Yes: 1495
Churn No: 4130
Churn Yes: 4130
Churn No: 4130

(4130, 4130)
```

```
###Oversample misses###

missDf = pd.concat([probdf, misses], ignore_index=True)
len(missDf)
```

```
6698
```

```
###Hybrid###

missedClass1 = misses[misses["Churn_Yes"] == 1]
overmiss = missedClass1.sample(n=(distribution[1]-distribution[0]), replace = Tr

overMissDf = pd.concat([probdf, overmiss], ignore_index=True)
dist(overMissDf)
```

```
Churn Yes: 4130
Churn No: 4130
```

Oversampled Datasets

Similar Probabilities, Different Classes

Method 1:

- Find a range of probabilities that that has both class0 and class1
- The sample units in that range have similar probabilities but different classes
- Issue: The range comprised of ~5000 units and the entire training set has ~5600 units. So, separating the 2 does not make a big difference

Method 2:

- Find a sample unit with the closest probability, but in class 0
- This way, the number of separated units is ~3000 out of ~5600

Similar Probabilities, Different Classes

- Took about 2-3 sec to run

-Should this be done with all training units in class 1or just missed units?

Similar Probability, Different Class

```
def findClosest(num):
    i = (class0['Prob'] - num).abs().idxmin()
    row = class0.loc[i]
    return row
```

```
class1 = probdf[probdf["Churn_Yes"] == 1].sort_values(by="Prob")
class0 = probdf[probdf["Churn_Yes"] == 0].sort_values(by="Prob")

SBD = pd.DataFrame()

import warnings
with warnings.catch_warnings():
    warnings.simplefilter(action='ignore', category=FutureWarning)
    for i in range(len(class1)):
        #print(len(SBD))
        row = class1.iloc[i]
        SBD = SBD.append(row)
        row2 = findClosest(row["Prob"])
        SBD = SBD.append(row2)
```

```
SBD[["Churn_Yes", "Prob", "Class"]].head(5)
```

	Churn_Yes	Prob	Class
268	1.0	0.004228	0.0
4280	0.0	0.004217	0.0
4819	1.0	0.005233	0.0
3553	0.0	0.005233	0.0
4386	1.0	0.006440	0.0

Hard to Predict

Hard to predict: High probability of class1, but class0 and vice versa

Easy to predict: Not hard to predict

The idea: The model was off by a lot with these sample units, so separating them might help

Expectation: It probably wont help the model since these units might simply be outliers.

There are not many in the hard to predict range

Hard/Easy To Predict

```
#Min and max prob
print("Min", max(float(class0["Prob"].head(1)), float(class1["Prob"].head(1))))
print("Max", min(float(class0["Prob"].tail(1)), float(class1["Prob"].tail(1))))
```

```
Min 0.004227954378193088
Max 0.8369896770916917
```

```
#Class 0 with high prob
C0HP = class0[(class0["Prob"]>.60)]

#Class 1 with low prob
C1LP = class1[(class1["Prob"]<.40)]

with warnings.catch_warnings():
    warnings.simplefilter(action='ignore', category=FutureWarning)
    HTP = pd.DataFrame().append(C0HP).append(C1LP)

print(len(HTP))

HTP[["Churn_Yes", "Prob", "Class"]].head(5)
```

```
743
```

	Churn_Yes	Prob	Class
2022	0	0.600133	1
5370	0	0.601239	1
5950	0	0.601686	1
1331	0	0.602272	1
273	0	0.602821	1

Hard to distinguish

Method:

- Use some measure to determine how difficult sample units are to classify
- Distance measures?

Work in progress

NN in Python

Can use *shuffle = False* to stop random sampling

```
Epoch 200/200
176/176 [==============================] - 0s 1ms/step - loss: 0.4504 - accur
acy: 0.7883 - recall_5: 0.4047
```

Neural Network

```
%%capture
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

# Define the model architecture
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=30))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
              metrics=['accuracy',tf.keras.metrics.Recall()])

# Train the model
model.fit(xtrain, ytrain, epochs=200, batch_size=32, shuffle=False)
```

```
pred = model.predict(xtest)
classPred = [0 if val < 0.5 else 1 for val in pred]
```

```
44/44 [==============================] - 0s 727us/step
```

```
evaluate(ytest, classPred)
```

```
Confusion Matrix:
[[971  62]
 [240 134]]

True Positive: 134
True Negative: 971
False Positive: 62
False Negative: 240

Accuracy: 0.7853589196872779
True-Positve Rate: 0.3582887700534759
```

Results: Oversample

```
Epoch 199/200
259/259 [==============================] - 0s 1ms/step - loss: 0.5340 - accur
acy: 0.7155 - recall_4: 0.7656
Epoch 200/200
259/259 [==============================] - 0s 1ms/step - loss: 0.5255 - accur
acy: 0.7156 - recall_4: 0.7751
44/44 [==============================] - 0s 1ms/step
Confusion Matrix:
[[133 900]
 [  2 372]]

True Positive: 372
True Negative: 133
False Positive: 900
False Negative: 2

Accuracy: 0.35891968727789625
True-Positve Rate: 0.9946524064171123
```

```
Confusion Matrix:
[[971  62]
 [240 134]]

True Positive: 134
True Negative: 971
False Positive: 62
False Negative: 240

Accuracy: 0.7853589196872779
True-Positve Rate: 0.3582887700534759
```

Results: Oversample Misses

```
Epoch 199/200
210/210 [==============================] - 0s 1ms/step - loss: 0.5677 - accur
acy: 0.6923 - recall_4: 0.1198
Epoch 200/200
210/210 [==============================] - 0s 1ms/step - loss: 0.5676 - accur
acy: 0.6927 - recall_4: 0.1268
44/44 [==============================] - 0s 999us/step
Confusion Matrix:
[[1031    2]
 [ 351   23]]

True Positive: 23
True Negative: 1031
False Positive: 2
False Negative: 351

Accuracy: 0.7491115849324804
True-Positve Rate: 0.06149732620320856
```

```
Confusion Matrix:
[[971  62]
 [240 134]]

True Positive: 134
True Negative: 971
False Positive: 62
False Negative: 240

Accuracy: 0.7853589196872779
True-Positve Rate: 0.3582887700534759
```

Results: Hybrid

```
Epoch 199/200
259/259 [==============================] - 0s 1ms/step - loss: 0.6299 - accur
acy: 0.6169 - recall_4: 0.7264
Epoch 200/200
259/259 [==============================] - 0s 1ms/step - loss: 0.6331 - accur
acy: 0.6105 - recall_4: 0.7249
44/44 [==============================] - 0s 756us/step
Confusion Matrix:
[[154 879]
 [  4 370]]

True Positive: 370
True Negative: 154
False Positive: 879
False Negative: 4

Accuracy: 0.3724235963041933
True-Positve Rate: 0.9893048128342246
```

```
Confusion Matrix:
[[971  62]
 [240 134]]

True Positive: 134
True Negative: 971
False Positive: 62
False Negative: 240

Accuracy: 0.7853589196872779
True-Positve Rate: 0.3582887700534759
```

Oversample Thoughts

Hybrid and oversample show clear signs of overfit since the recall and accuracy on training data is good.

Overall, this did not work

*note: I did not give the NN model the “prob” variable with Logistic probabilities

Results: Similar but Different

```
Epoch 199/200
176/176 [==============================] - 0s 1ms/step - loss: 0.5086 - accur
acy: 0.7744 - recall_5: 0.2297
Epoch 200/200
176/176 [==============================] - 0s 1ms/step - loss: 0.4950 - accur
acy: 0.7794 - recall_5: 0.2827
44/44 [==============================] - 0s 846us/step
Confusion Matrix:
[[1033    0]
 [ 372    2]]

True Positive: 2
True Negative: 1033
False Positive: 0
False Negative: 372

Accuracy: 0.7356076759061834
True-Positve Rate: 0.0053475935828877
```

```
Confusion Matrix:
[[971  62]
 [240 134]]

True Positive: 134
True Negative: 971
False Positive: 62
False Negative: 240

Accuracy: 0.7853589196872779
True-Positve Rate: 0.3582887700534759
```

Results: Hard to Predict

```
Epoch 199/200
176/176 [==============================] - 0s 1ms/step - loss: 0.4659 - accuracy: 0.7557 - recall_5: 0.2021
Epoch 200/200
176/176 [==============================] - 0s 1ms/step - loss: 0.5230 - accuracy: 0.7582 - recall_5: 0.2015
44/44 [==============================] - 0s 821us/step
Confusion Matrix:
[[1019   14]
 [ 313   61]]

True Positive: 61
True Negative: 1019
False Positive: 14
False Negative: 313

Accuracy: 0.767590618336887
True-Positve Rate: 0.16310160427807488
```

```
Confusion Matrix:
[[971  62]
 [240 134]]

True Positive: 134
True Negative: 971
False Positive: 62
False Negative: 240

Accuracy: 0.7853589196872779
True-Positve Rate: 0.3582887700534759
```

Week 2

Goals

1. Shuffle
2. Thresholds
3. ROC curve and AUC values
4. Epoch graph
    1. Batch graph
5. Error Weights

1. Shuffle

https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

`shuffle`	Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). This argument is ignored when x is a generator or an object of tf.data.Dataset. 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not None.

https://github.com/keras-team/keras/blob/master/keras/engine/training_arrays_v1.py

```
376                 index_array = np.arange(num_samples_or_steps)
377                 if shuffle == "batch":
378                     index_array = training_utils_v1.batch_shuffle(
379                         index_array, batch_size
380                     )
381                 elif shuffle:
382                     np.random.shuffle(index_array)
```

```
class printeverybatch(tf.keras.Model):
    def train_step(self, data):
        x, y = data
        tf.print('new batch:')
        tf.print(x,summarize=-1)
        tf.print(y,summarize=-1)
        return super().train_step(data)

tf.random.set_seed(88883)
np.random.seed(88883)

inputs=tf.keras.Input((5,))
model=printeverybatch(inputs,tf.keras.layers.Dense(1)(inputs))

print("x :", x)
print("y :", y)
print("\n")
x=np.random.rand(4,5)
y=np.random.rand(4,1)
model.compile(loss=tf.keras.losses.MeanSquaredError(),
              optimizer=tf.keras.optimizers.SGD())
model.fit(x,y,batch_size=2,verbose=2,epochs=2)
```

https://stackoverflow.com/questions/68115889/how-to-print-every-x-y-pair-used-during-a-tensorflow-training-model-fitx-y

Further Investigation

This code modifies the train_step function so that it prints the x and y values in each batch

```
x : [[0.53581302 0.29821178 0.66917816 0.98964157 0.38394524]
 [0.4942365  0.96995748 0.50016853 0.53725342 0.76896955]
 [0.22719473 0.27151703 0.27261427 0.57769282 0.1005021 ]
 [0.24324105 0.05896128 0.78992328 0.71833981 0.06964264]]
y : [[0.27548476]
 [0.20291323]
 [0.39257899]
 [0.49743296]]


Epoch 1/2
new batch:
[[0.227194726 0.271517038 0.27261427 0.577692807 0.100502096]
 [0.535813034 0.298211783 0.669178188 0.989641547 0.383945227]]
[[0.392579]
 [0.275484771]]
new batch:
[[0.494236499 0.969957471 0.500168502 0.537253439 0.768969536]
 [0.243241057 0.0589612834 0.78992331 0.718339801 0.0696426481]]
[[0.20291324]
 [0.497432947]]
2/2 - 0s - loss: 0.3503 - 182ms/epoch - 91ms/step
Epoch 2/2
new batch:
[[0.535813034 0.298211783 0.669178188 0.989641547 0.383945227]
 [0.243241057 0.0589612834 0.78992331 0.718339801 0.0696426481]]
[[0.275484771]
 [0.497432947]]
new batch:
[[0.494236499 0.969957471 0.500168502 0.537253439 0.768969536]
 [0.227194726 0.271517038 0.27261427 0.577692807 0.100502096]]
[[0.20291324]
 [0.392579]]
2/2 - 0s - loss: 0.2933 - 17ms/epoch - 8ms/step
```

Shuffle = True

```
model.fit(x,y,batch_size=2,verbose=2,epochs=2, shuffle=False)
```

```
x : [[0.53581302 0.29821178 0.66917816 0.98964157 0.38394524]
 [0.4942365  0.96995748 0.50016853 0.53725342 0.76896955]
 [0.22719473 0.27151703 0.27261427 0.57769282 0.1005021 ]
 [0.24324105 0.05896128 0.78992328 0.71833981 0.06964264]]
y : [[0.27548476]
 [0.20291323]
 [0.39257899]
 [0.49743296]]


Epoch 1/2
new batch:
[[0.535813034 0.298211783 0.669178188 0.989641547 0.383945227]
 [0.494236499 0.969957471 0.500168502 0.537253439 0.768969536]]
[[0.275484771]
 [0.20291324]]
new batch:
[[0.227194726 0.271517038 0.27261427 0.577692807 0.100502096]
 [0.243241057 0.0589612834 0.78992331 0.718339801 0.0696426481]]
[[0.392579]
 [0.497432947]]
2/2 - 0s - loss: 1.6938 - 182ms/epoch - 91ms/step
Epoch 2/2
new batch:
[[0.535813034 0.298211783 0.669178188 0.989641547 0.383945227]
 [0.494236499 0.969957471 0.500168502 0.537253439 0.768969536]]
[[0.275484771]
 [0.20291324]]
new batch:
[[0.227194726 0.271517038 0.27261427 0.577692807 0.100502096]
 [0.243241057 0.0589612834 0.78992331 0.718339801 0.0696426481]]
[[0.392579]
 [0.497432947]]
2/2 - 0s - loss: 1.4107 - 17ms/epoch - 9ms/step
```

Perfectly in order

Shuffle = False

Apply to the Data

```
class printeverybatch(tf.keras.Model):
    def train_step(self, data):
        x, y = data
        tf.print('new batch:')
        tf.print(x,summarize=-1)
        tf.print(y,summarize=-1)
        return super().train_step(data)

inputs=tf.keras.Input((30,))
model=printeverybatch(inputs,tf.keras.layers.Dense(1)(inputs))

x=xtrain
y=ytrain
model.compile(loss=tf.keras.losses.MeanSquaredError(),
              optimizer=tf.keras.optimizers.SGD())
model.fit(x,y,batch_size=2,verbose=2,epochs=2, shuffle=False)
```

Experimenting With the Dataset

```
In [37]: xtrain.head(5)
```

Out[37]:

		tenure	MonthlyCharges	TotalCharges	gender_Male	SeniorCitizen_1	Partner_Yes	Dependents_Yes	PhoneService_Yes	MultipleLines_No phone service	MultipleLines_Ye
1	6030	43	49.05	2076.20	0	0	0	0	0	1	
	3410	3	53.40	188.70	1	0	0	0	1	0	
	5483	55	77.75	4458.15	0	0	1	0	1	0	
	5524	45	54.65	2553.70	1	0	1	1	1	0	
	6337	55	100.90	5448.60	0	0	1	1	1	0	

```
Epoch 2/2
new batch:
1[[43 49.05 2076.2 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0]
 [3 53.4 188.7 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0]]
[[1]
 [1]]
new batch:
[[55 77.75 4458.15 0 0 1 0 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0]
 [45 54.65 2553.7 1 0 1 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0]]
[[1]
 [0]]
new batch:
[[55 100.9 5448.6 0 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1]
 [5 45.7 198 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0]]
[[0]
 [1]]
```

```
model.fit(x,y,batch_size=2,verbose=2,epochs=2, shuffle=False)
```

```
Epoch 1/2
new batch:
1[[43 49.05 2076.2 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0]
 [3 53.4 188.7 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0]]
[[1]
 [1]]
new batch:
[[55 77.75 4458.15 0 0 1 0 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0]
 [45 54.65 2553.7 1 0 1 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0]]
[[1]
 [0]]
new batch:
[[55 100.9 5448.6 0 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1]
 [5 45.7 198 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0]]
[[0]
 [1]]
new batch:
[[59 99.5 5961.1 1 0 1 0 1 0 1 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 1 0 0 0]
 [12 96 1062.1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1 0 1 0]]
```

2. Thresholds

Multiplied acc and recall to try to consider both

```
def thresh(pred, ytest):
    bestacc = 0
    besttp = 0
    bestthresh = 0
    accList = []
    tpList = []
    threshlist = []
    for i in range(1,100):
        classPred = [0 if val < (i/100) else 1 for val in pred]
        accuracy = accuracy_score(ytest, classPred)
        tpr = recall_score(ytest, classPred)
        accList.append(accuracy)
        tpList.append(tpr)
        threshlist.append(i)
        if ((accuracy * tpr)>(bestacc * besttp)):
            bestacc= accuracy
            besttp = tpr
            bestthresh=(i/100)

    plt.plot(threshlist, accList, 'b', label='Accuracy')
    plt.plot(threshlist, tpList, 'r', label='Recall')
    plt.title('Threshold Graph')
    plt.xlabel('Threshold')
    plt.ylabel('Metric Value')
    plt.legend()
    plt.show()
    print("Best Threshold:", bestthresh)
    print("Accuracy:", bestacc)
    print("Recall:", besttp)

    return bestthresh
```

3. ROC and AUC

https://www.statology.org/plot-roc-curve-python/

```
def plotROC(prob, ytest):
    fpr, tpr, _ = metrics.roc_curve(ytest,  prob)
    auc = metrics.roc_auc_score(ytest, prob)
    #create ROC curve
    plt.plot(fpr,tpr,label="AUC="+str(auc))
    plt.title('ROC and AUC')
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.legend(loc=4)
    plt.show()
```

4. Epoch Graph

https://stackoverflow.com/questions/41908379/keras-plot-training-validation-and-test-set-accuracy

```
def plotNN(history):
    # Extract accuracy and TPR values from the training histor
    accuracy = history.history['accuracy']
    val_accuracy = history.history['val_accuracy']
    tpr = history.history['recall_2']
    val_tpr = history.history['val_recall_2']
    epochs = range(1, len(accuracy) + 1)

    # Plot the accuracy values
    plt.plot(epochs, accuracy, 'b', label='Training Accuracy')
    plt.plot(epochs, val_accuracy, 'r', label='Validation Accu

    # Plot the TPR values
    plt.plot(epochs, tpr, 'g', label='Training TPR')
    plt.plot(epochs, val_tpr, 'm', label='Validation TPR')

    plt.title('Training and Validation Metrics')
    plt.xlabel('Epochs')
    plt.ylabel('Metric Value')
    plt.legend()
    plt.show()
```

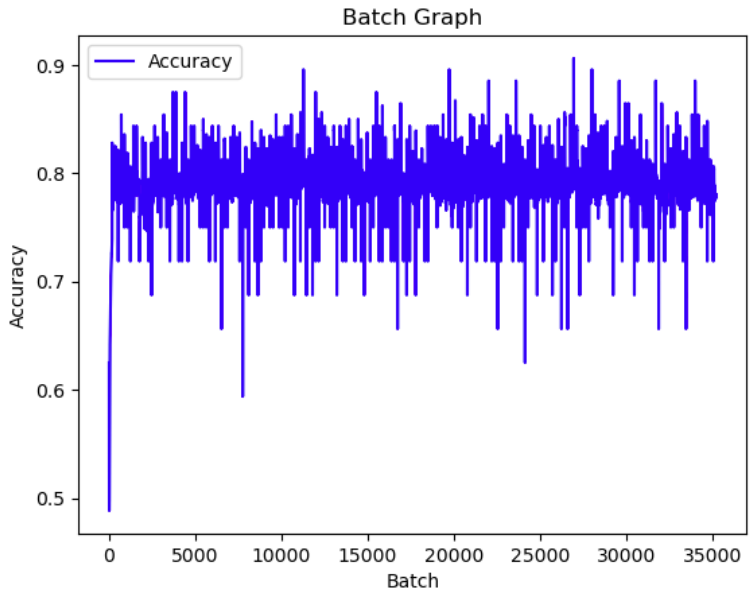
4.5: Batch Graph

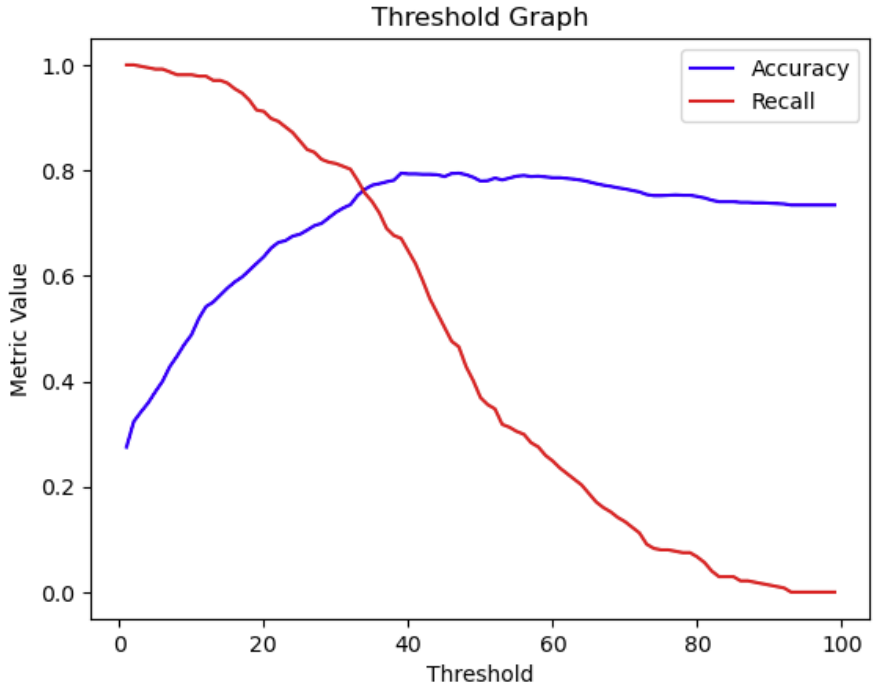
https://stackoverflow.com/questions/66394598/how-to-get-history-over-one-epoch-after-every-batch

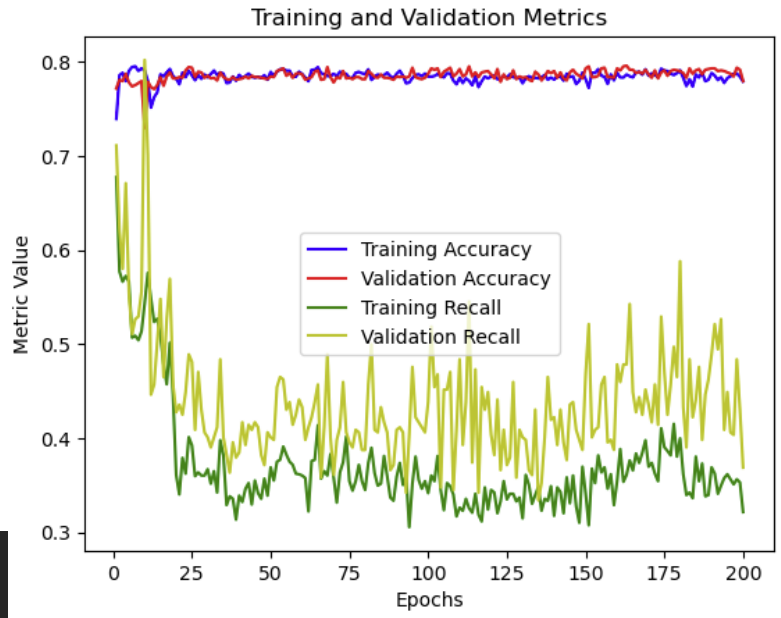
- Accuracy after each batch (updates gradient)
- Useful?
- Required larger batches or less epoch to make graph interpretable

```
#accuracy after each batch
class BCP(tf.keras.callbacks.Callback):
    batch_accuracy = [] # accuracy at given batch
    batch_loss = [] # loss at given batch
    def __init__(self):
        super(BCP,self).__init__()
    def on_train_batch_end(self, batch, logs=None):
        BCP.batch_accuracy.append(logs.get('accuracy'))
        BCP.batch_loss.append(logs.get('loss'))
```

Results: Regular Dataset (200/32)

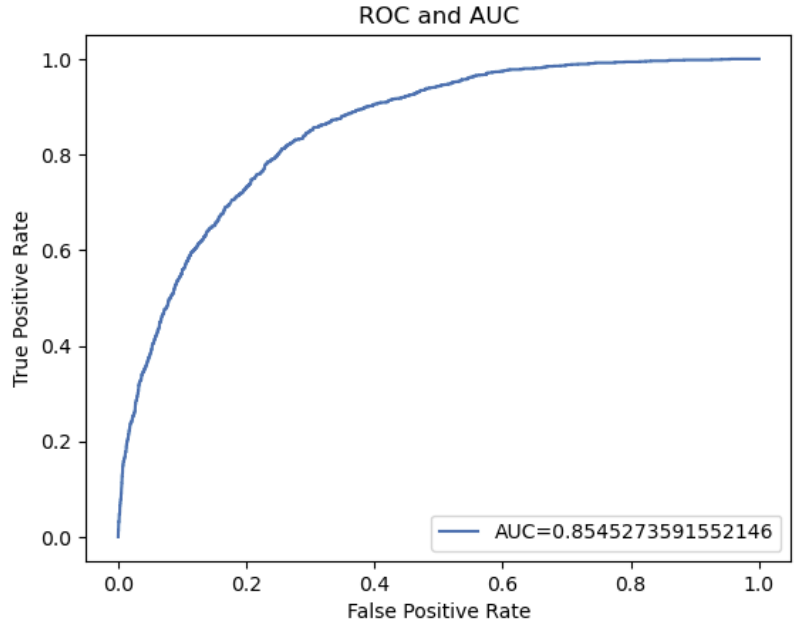




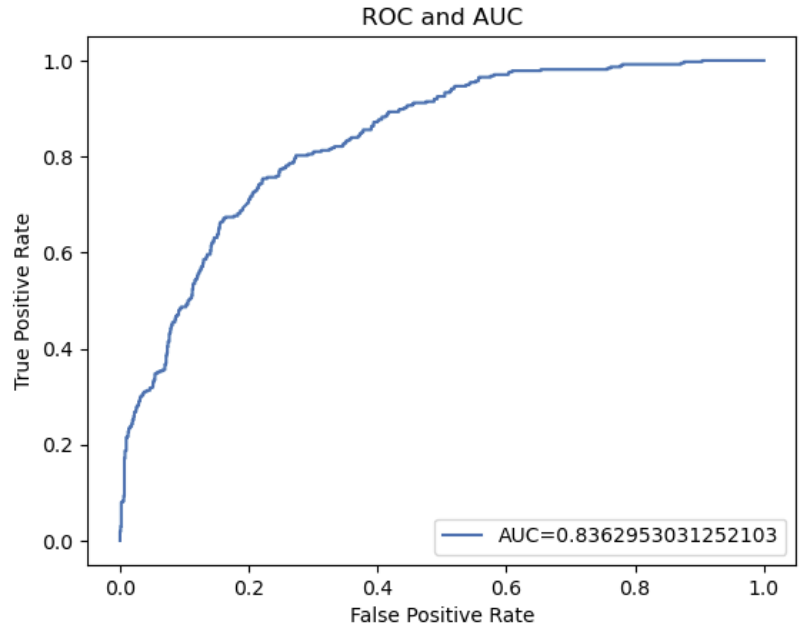



```
Best Threshold: 0.22
Accuracy: 0.6631130063965884
Recall: 0.893048128342246
Confusion Matrix:
[[599 434]
 [ 40 334]]

True Positive: 334
True Negative: 599
False Positive: 434
False Negative: 40

Accuracy: 0.6631130063965884
True-Positve Rate: 0.893048128342246
```

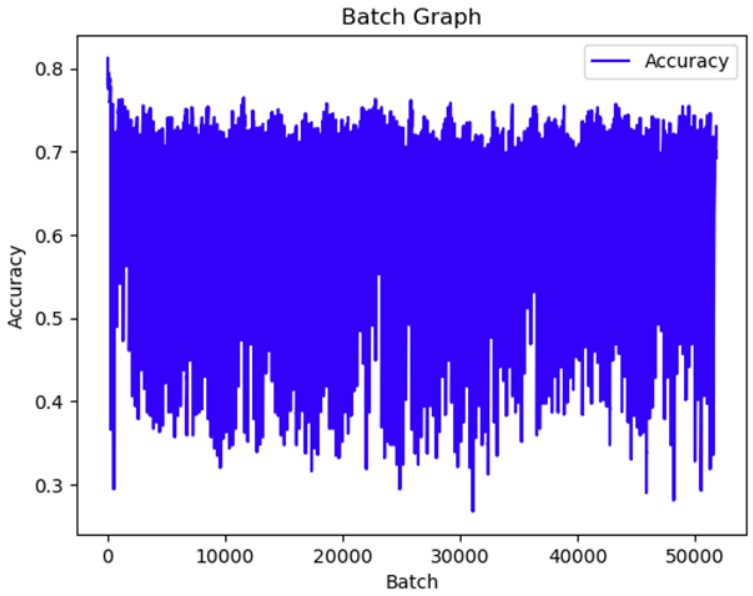
```
Training ROC:
176/176 [==============================] - 0s 808us/step
```

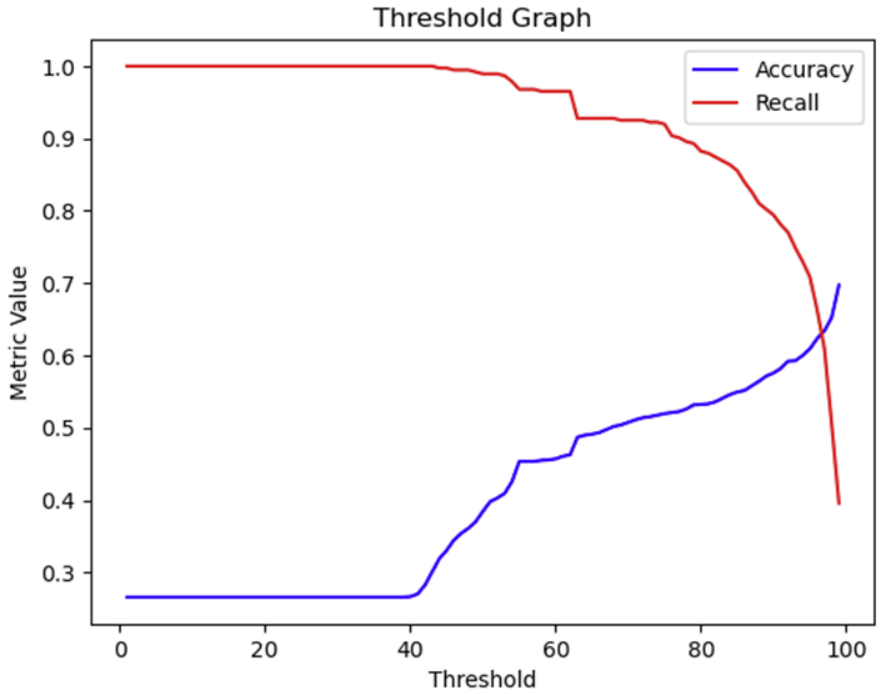



```
44/44 [==============================] - 0s 963us/step
Test ROC:
```

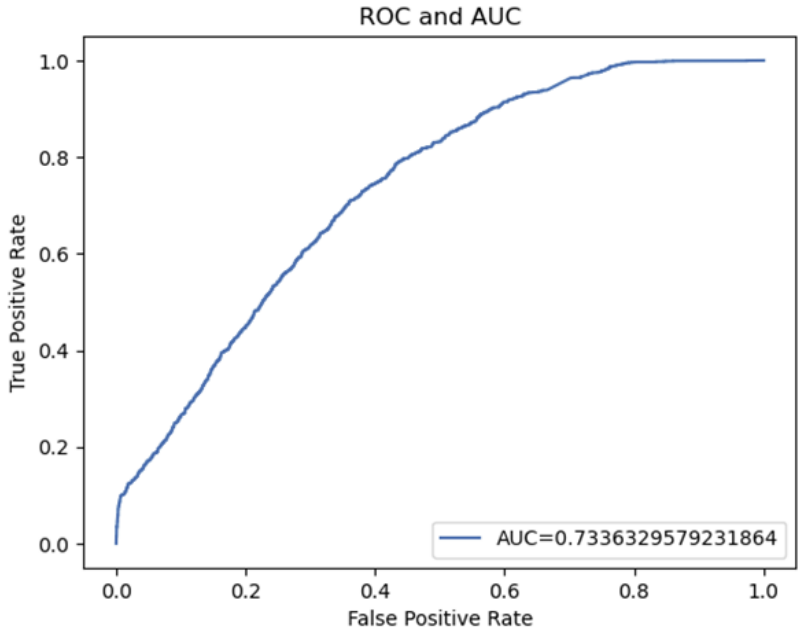


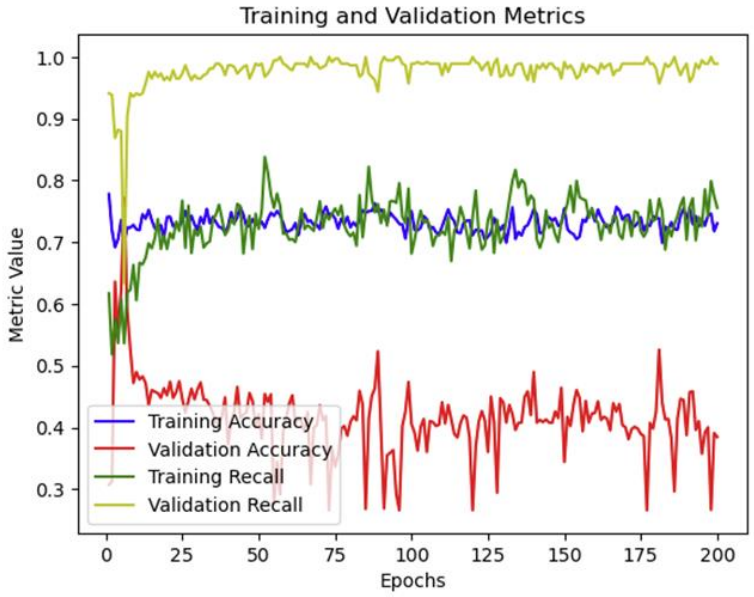
Results: Oversample





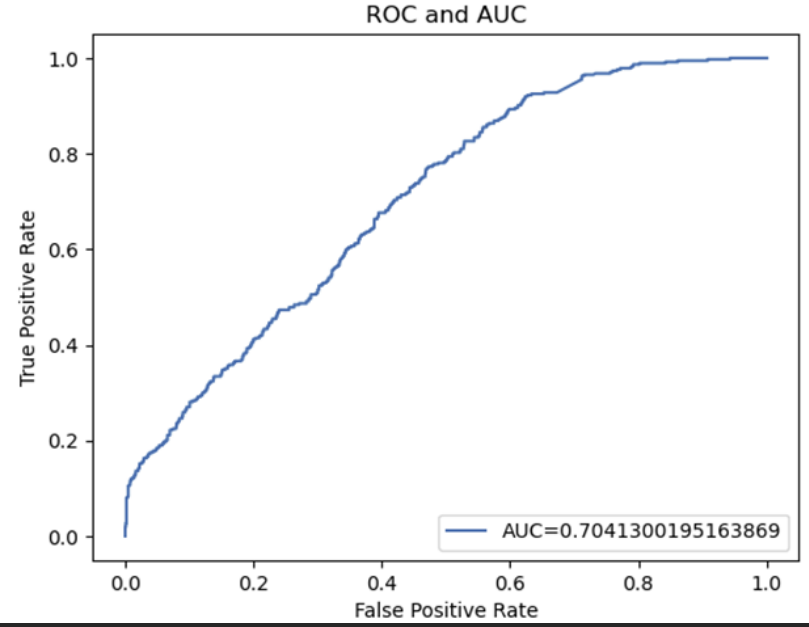

```
Training ROC:
259/259 [==============================] - 0s 745us/step
```



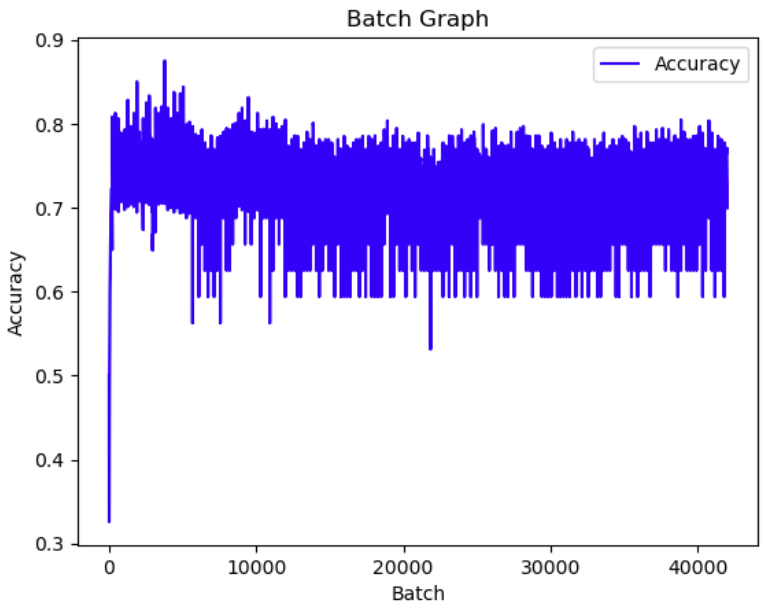


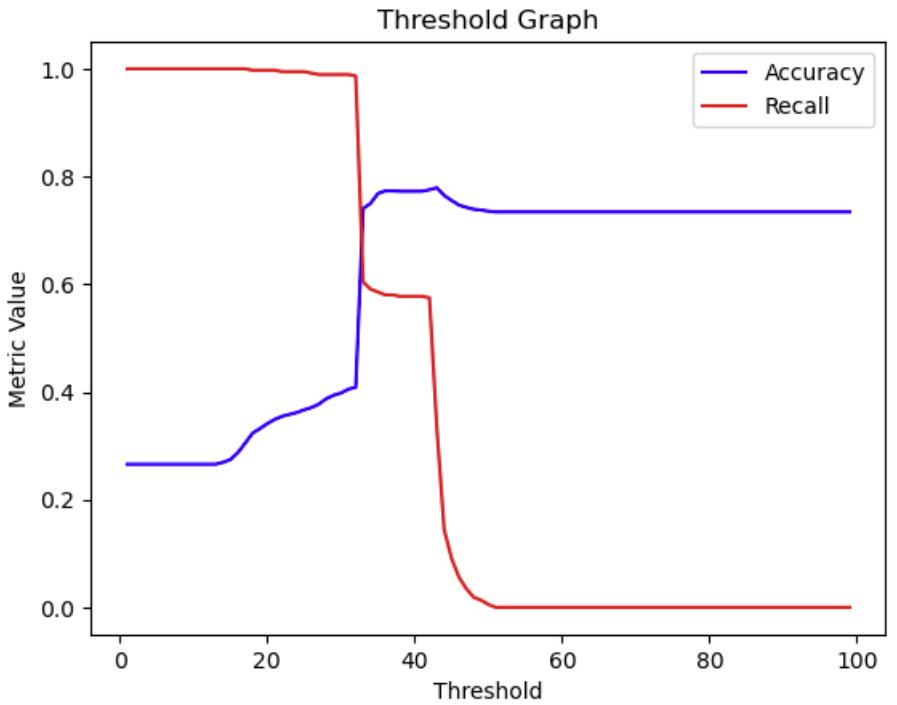

```
Best Threshold: 0.75
Accuracy: 0.51954513148543
Recall: 0.9197860962566845
Confusion Matrix:
[[387 646]
 [ 30 344]]

True Positive: 344
True Negative: 387
False Positive: 646
False Negative: 30

Accuracy: 0.51954513148543
True-Positve Rate: 0.9197860962566845
```

```
44/44 [==============================] - 0s 573us/step
Test ROC:
```

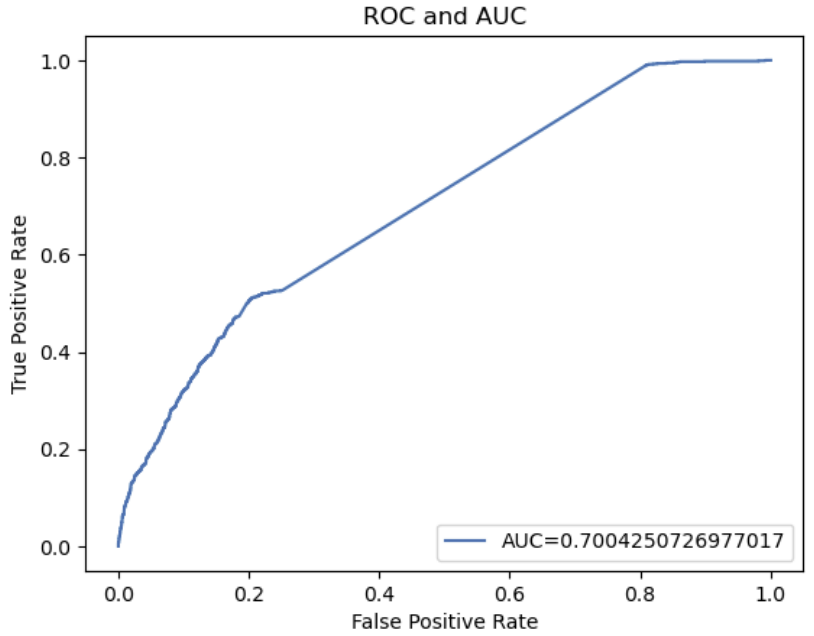


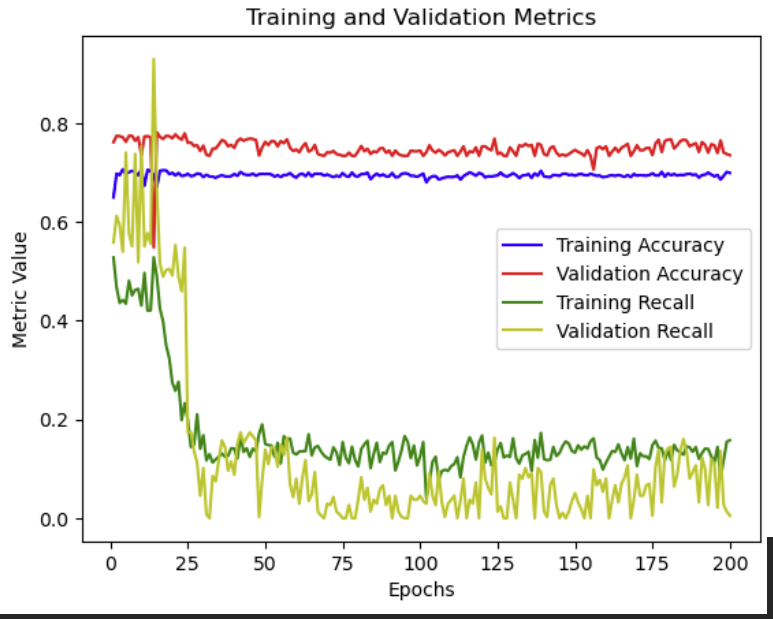
Results: Oversample Misses





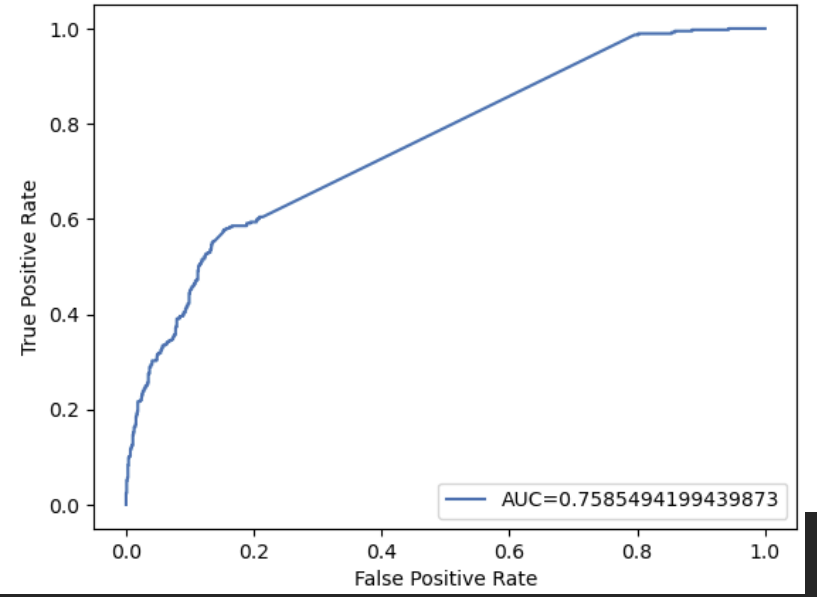

```
Training ROC:
210/210 [==============================] - 0s 745us/step
```



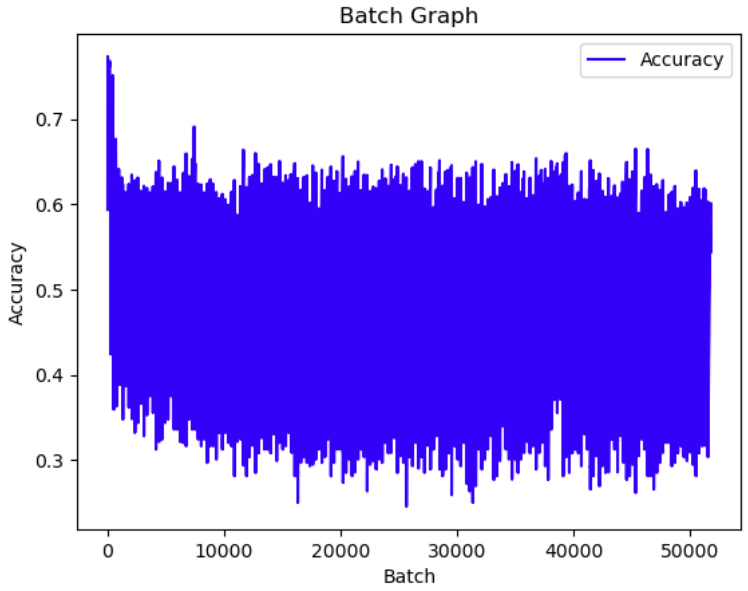


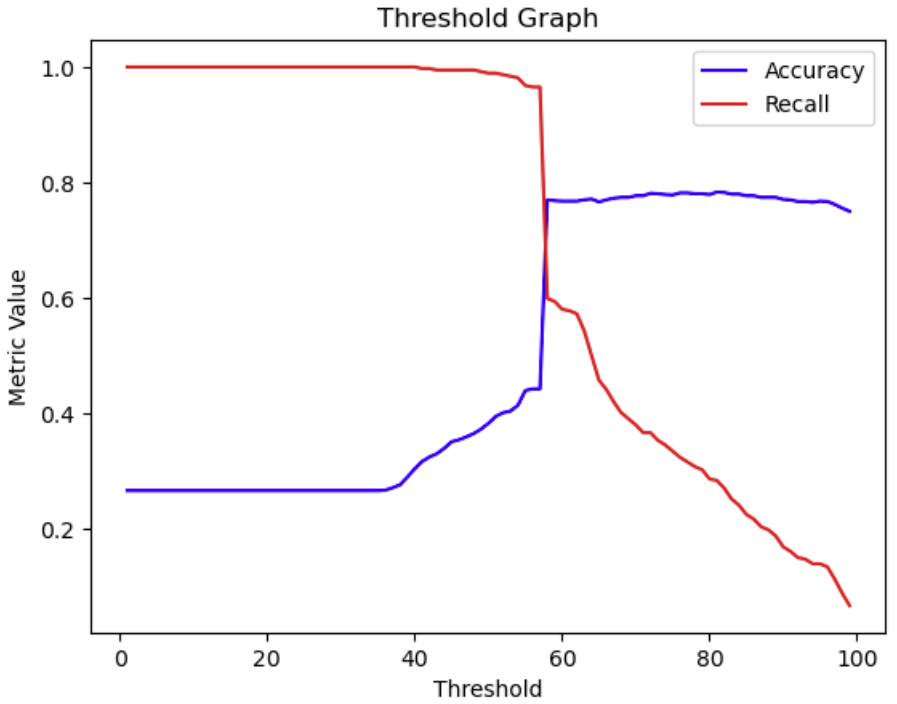

```
Best Threshold: 0.35
Accuracy: 0.7683013503909026
Recall: 0.5855614973262032
Confusion Matrix:
[[862 171]
 [155 219]]

True Positive: 219
True Negative: 862
False Positive: 171
False Negative: 155

Accuracy: 0.7683013503909026
True-Positve Rate: 0.5855614973262032
```

```
44/44 [==============================] - 0s 954us/step
Test ROC:
```

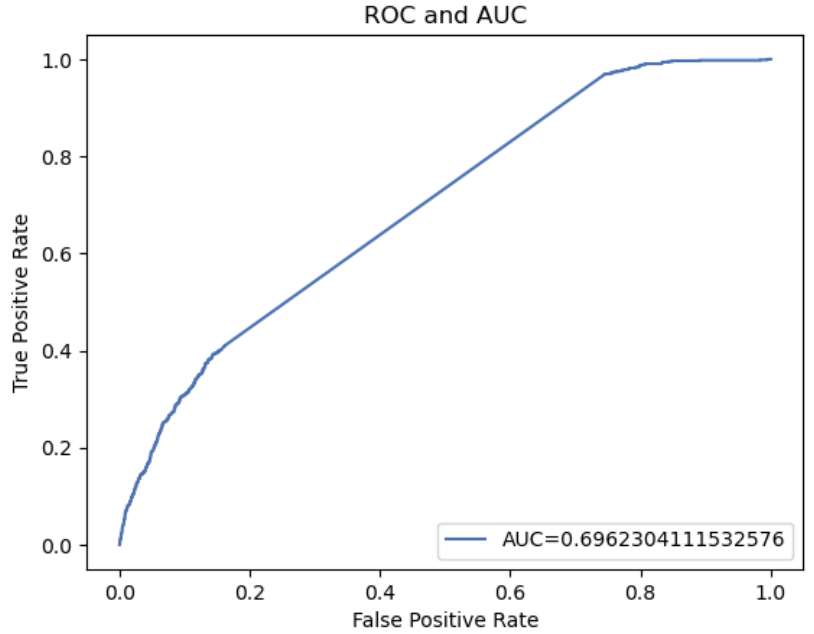


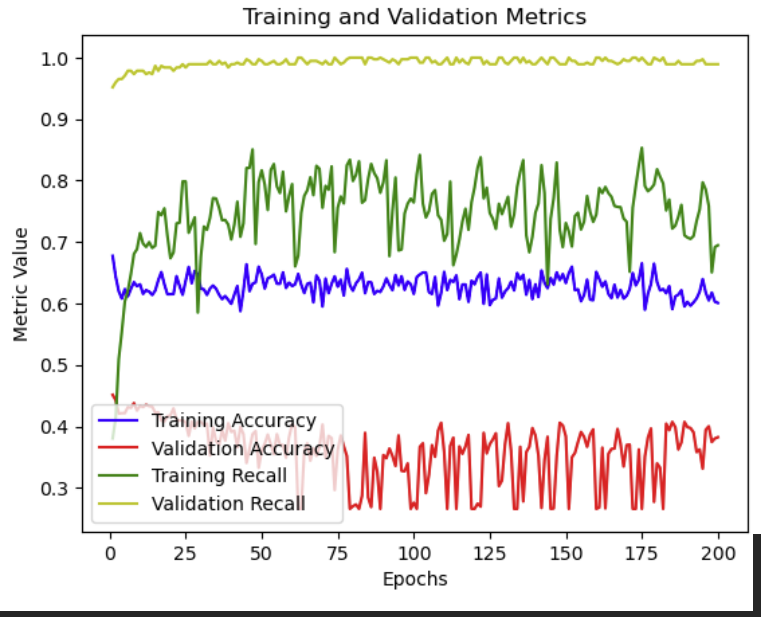
Results: Hybrid Oversample





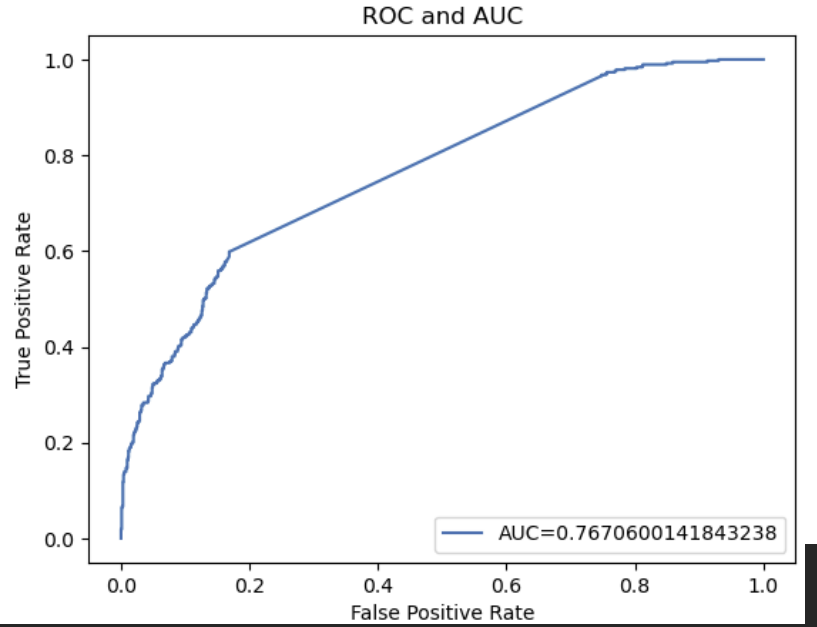

```
Training ROC:
259/259 [==============================] - 0s 790us/step
```



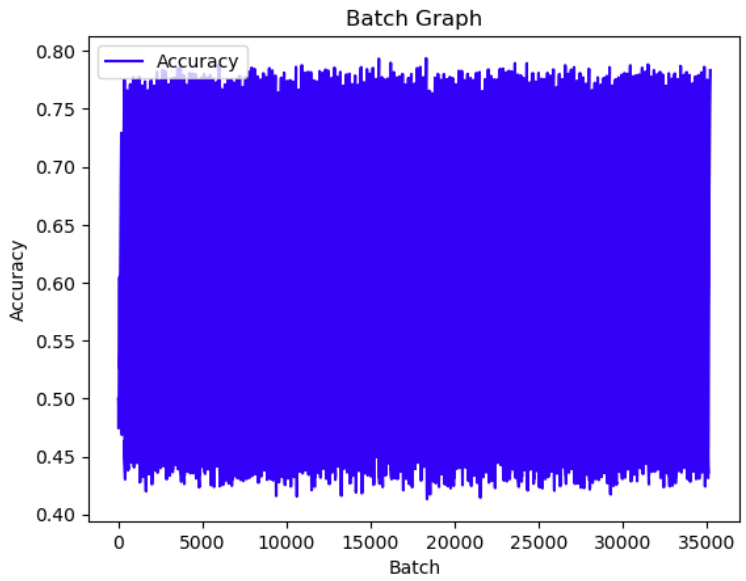


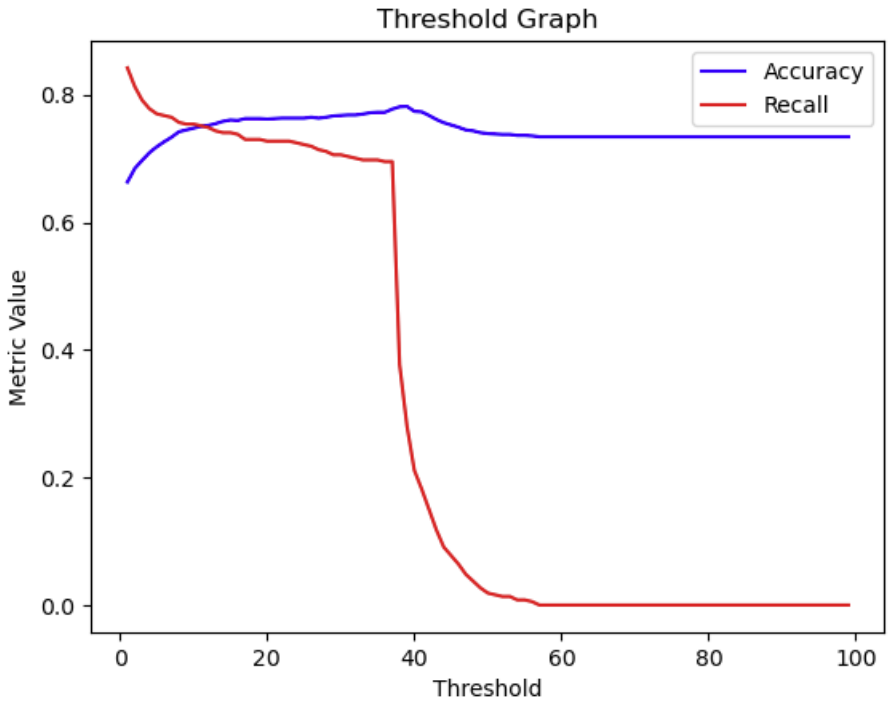

```
Best Threshold: 0.58
Accuracy: 0.7697228144989339
Recall: 0.5989304812834224
Confusion Matrix:
[[859 174]
 [150 224]]

True Positive: 224
True Negative: 859
False Positive: 174
False Negative: 150

Accuracy: 0.7697228144989339
True-Positve Rate: 0.5989304812834224
```

```
44/44 [==============================] - 0s 736us/step
Test ROC:
```

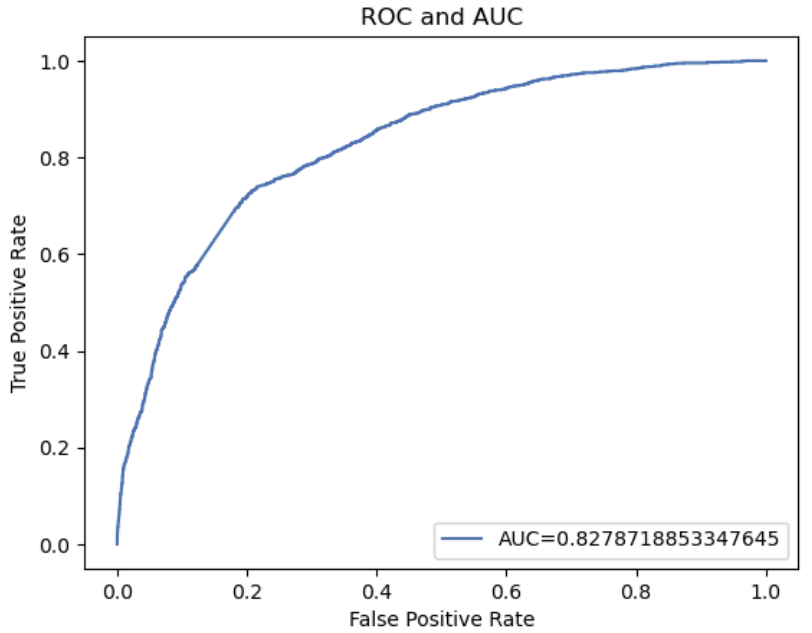


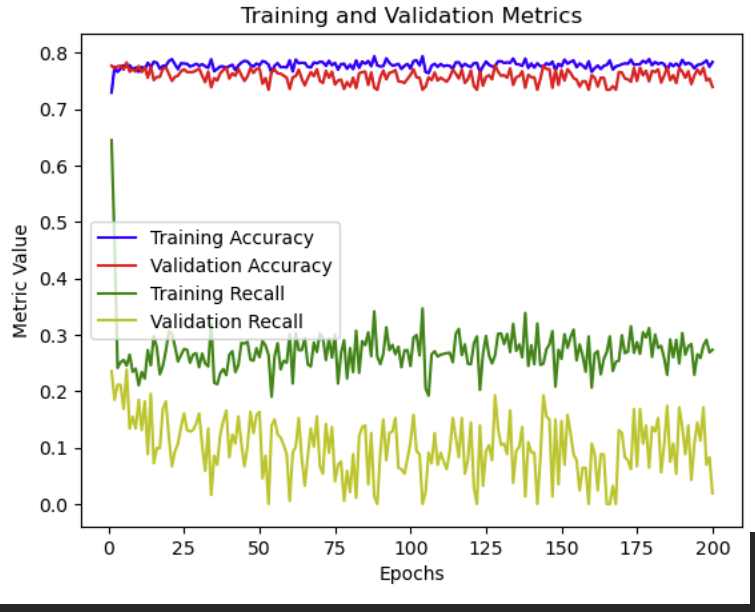
Results: Similar but different





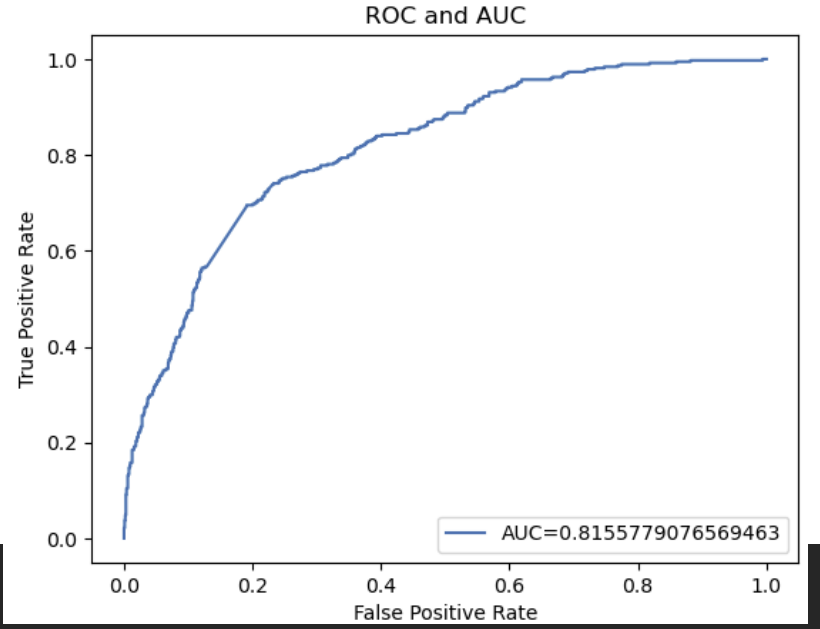

```
Training ROC:
176/176 [==============================] - 0s 766us/step
```



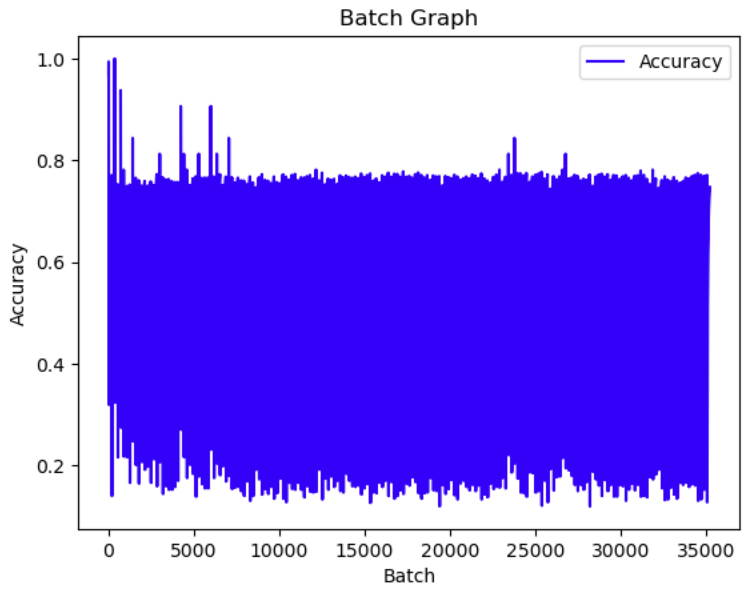


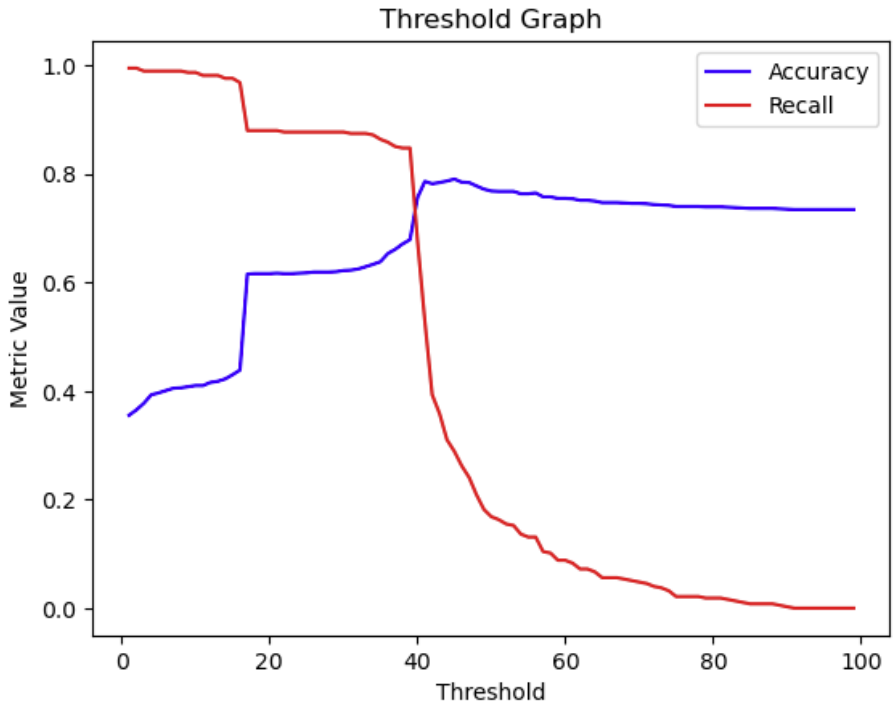

```
Best Threshold: 0.11
Accuracy: 0.7505330490405118
Recall: 0.7513368983957219
Confusion Matrix:
[[775 258]
 [ 93 281]]

True Positive: 281
True Negative: 775
False Positive: 258
False Negative: 93

Accuracy: 0.7505330490405118
True-Positve Rate: 0.7513368983957219
```

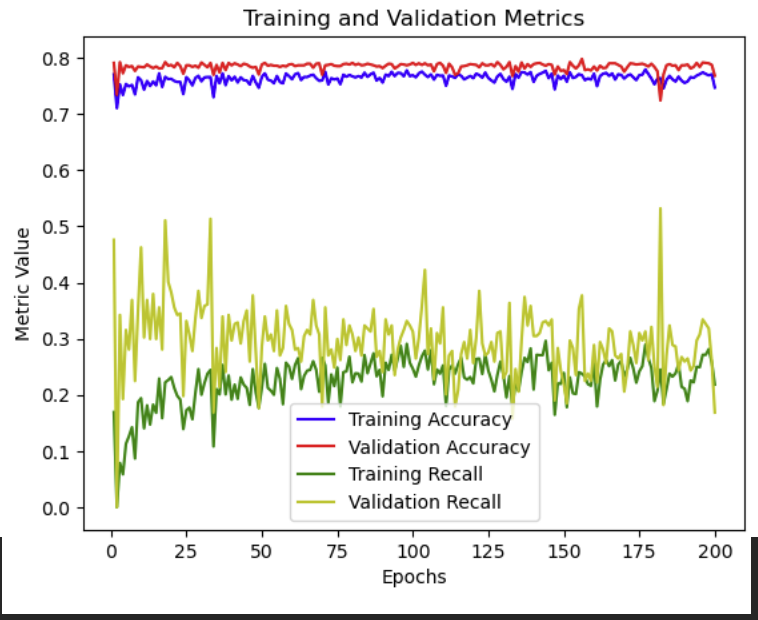
```
44/44 [==============================] - 0s 680us/step
Test ROC:
```



Results: Hard to predict

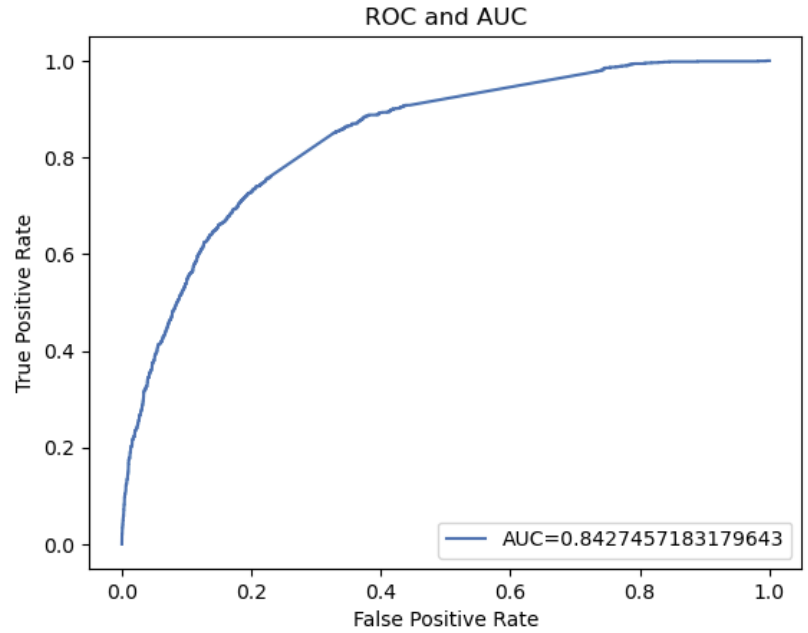




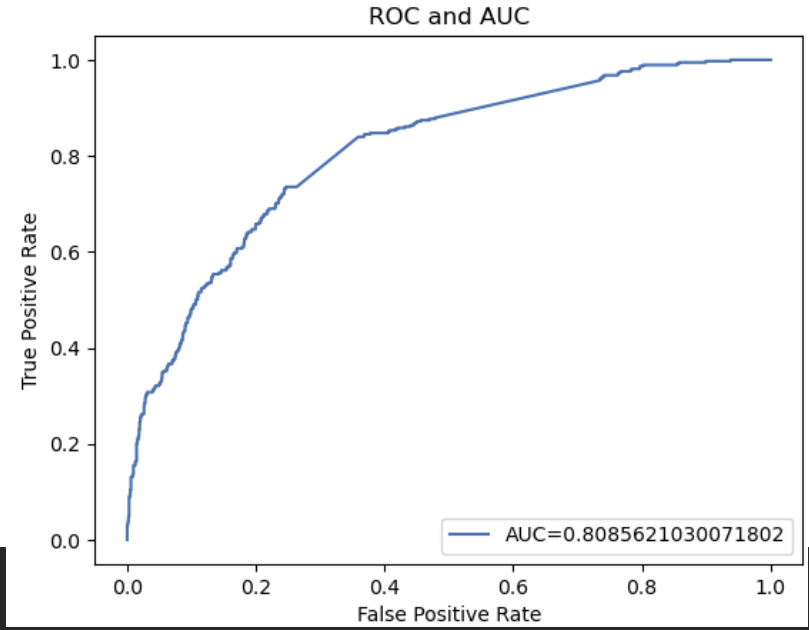



```
Best Threshold: 0.39
Accuracy: 0.6787491115849325
Recall: 0.8475935828877005
Confusion Matrix:
[[638 395]
 [ 57 317]]

True Positive: 317
True Negative: 638
False Positive: 395
False Negative: 57

Accuracy: 0.6787491115849325
True-Positve Rate: 0.8475935828877005
```

```
Training ROC:
176/176 [==============================] - 0s 663us/step
```




```
44/44 [==============================] - 0s 724us/step
Test ROC:
```



Conclusions

- Hard to predict and Similar but Different seem to improve the model
  - Showed signs of overfitting (high training recall and accuracy)

Easy First

```
Best Threshold: 0.39
Accuracy: 0.6787491115849325
Recall: 0.8475935828877005
Confusion Matrix:
[[638 395]
 [ 57 317]]

True Positive: 317
True Negative: 638
False Positive: 395
False Negative: 57

Accuracy: 0.6787491115849325
True-Positve Rate: 0.8475935828877005
```

Hard to Predict

```
Best Threshold: 0.4
Accuracy: 0.660270078180526
Recall: 0.8048128342245989
Confusion Matrix:
[[628 405]
 [ 73 301]]

True Positive: 301
True Negative: 628
False Positive: 405
False Negative: 73

Accuracy: 0.660270078180526
True-Positve Rate: 0.8048128342245989
```

```
Best Threshold: 0.11
Accuracy: 0.7505330490405118
Recall: 0.7513368983957219
Confusion Matrix:
[[775 258]
 [ 93 281]]

True Positive: 281
True Negative: 775
False Positive: 258
False Negative: 93

Accuracy: 0.7505330490405118
True-Positve Rate: 0.7513368983957219
```

Similar but Different

```
Best Threshold: 0.31
Accuracy: 0.6624022743425728
Recall: 0.839572192513369
Confusion Matrix:
[[618 415]
 [ 60 314]]

True Positive: 314
True Negative: 618
False Positive: 415
False Negative: 60

Accuracy: 0.6624022743425728
True-Positve Rate: 0.839572192513369
```

Feeding difficult sample units first seems to be more effective

To shuffle or not to shuffle

- By lowering the number of epoch and increasing batch size, we can create a better graph of the accuracy after each batch

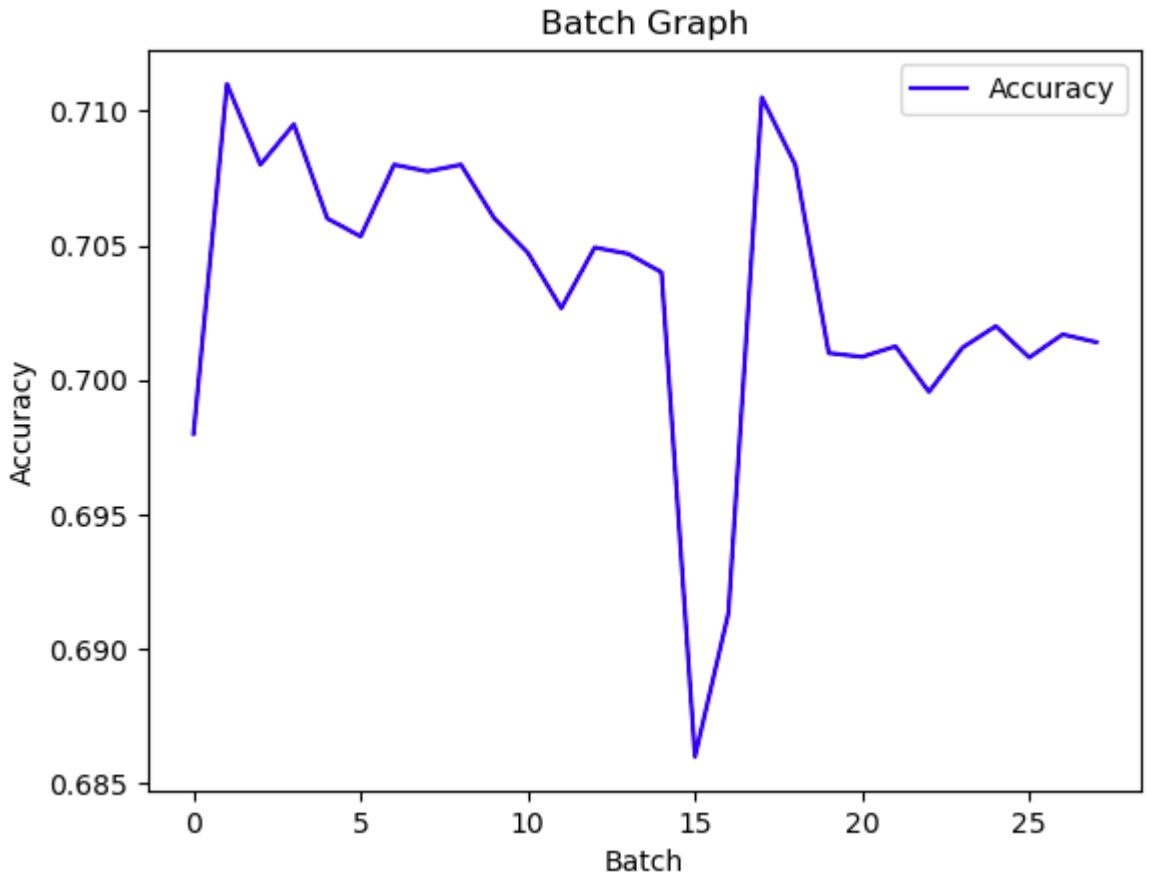
- This will be performed with 3 data sets: oversample misses, similar but different, and hard to predict

- There will be 2 epochs, and batches will be 500 sample units

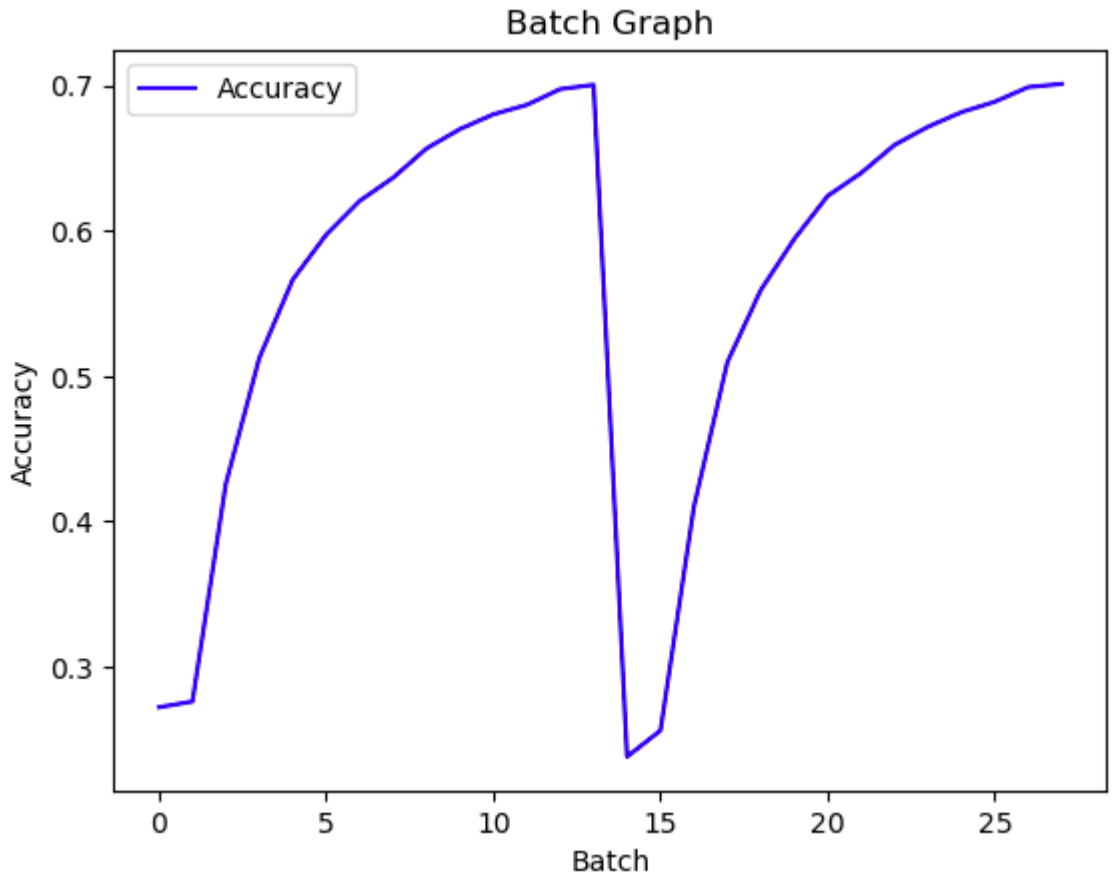
oversample misses

```
total: 6698
total misses: 1073
```

Shuffle



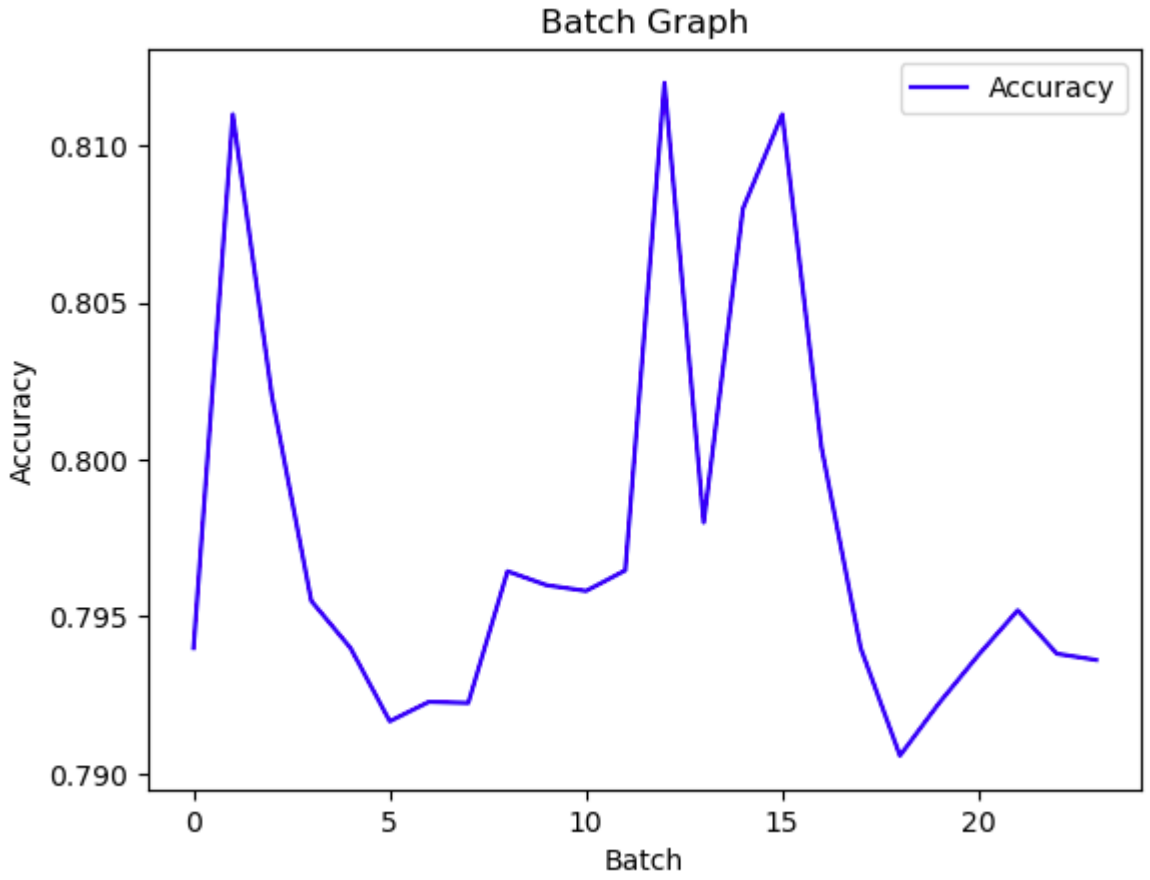
Don't Shuffle



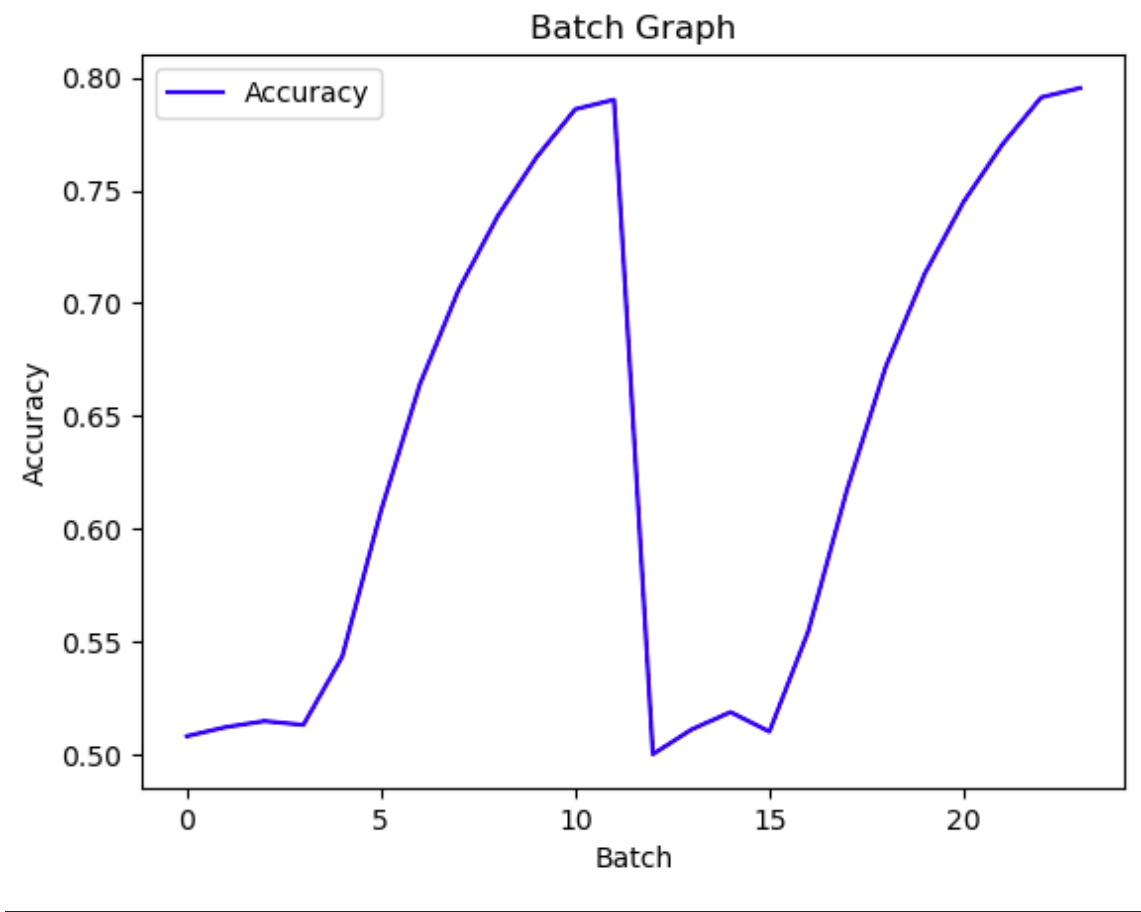
similar but different

```
total: 5611
SBD: 2990
```

Shuffle



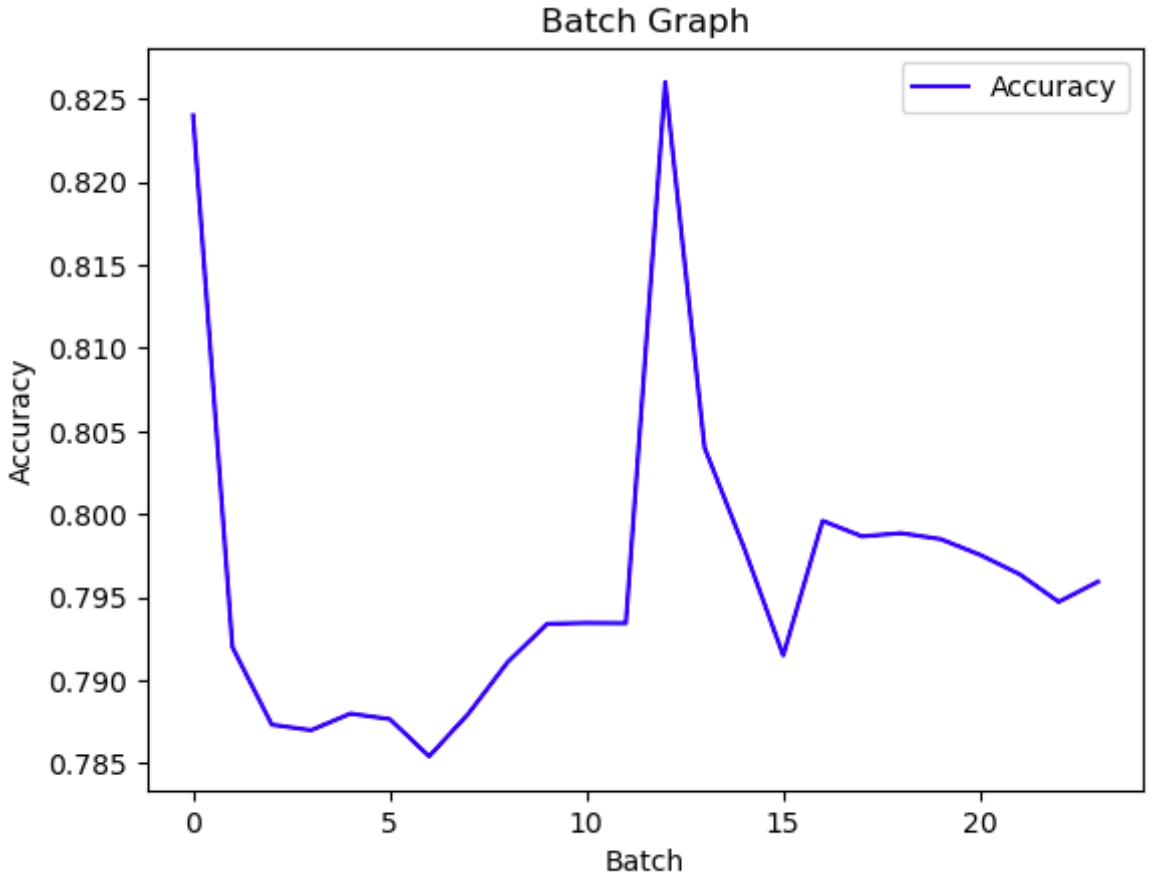
Don’t Shuffle



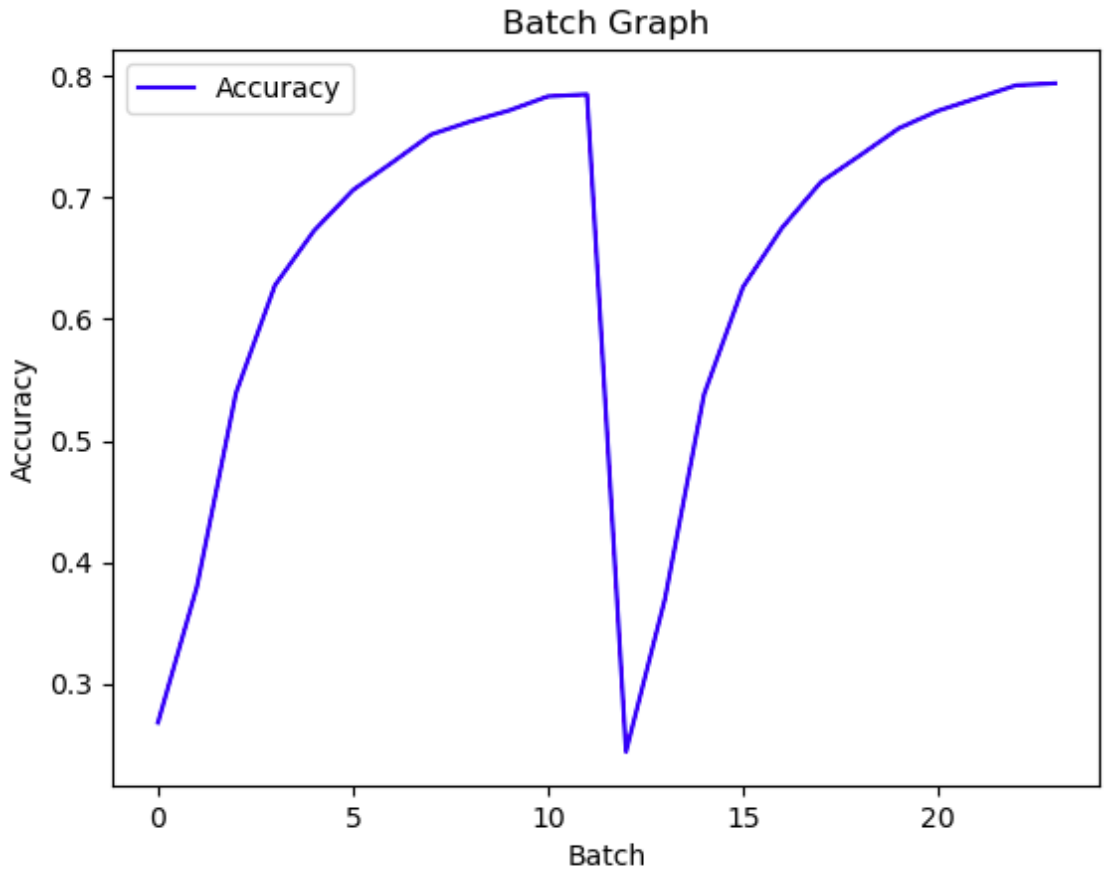
hard to predict

```
total: 5611
HTP: 743
```

Shuffle



Don’t Shuffle



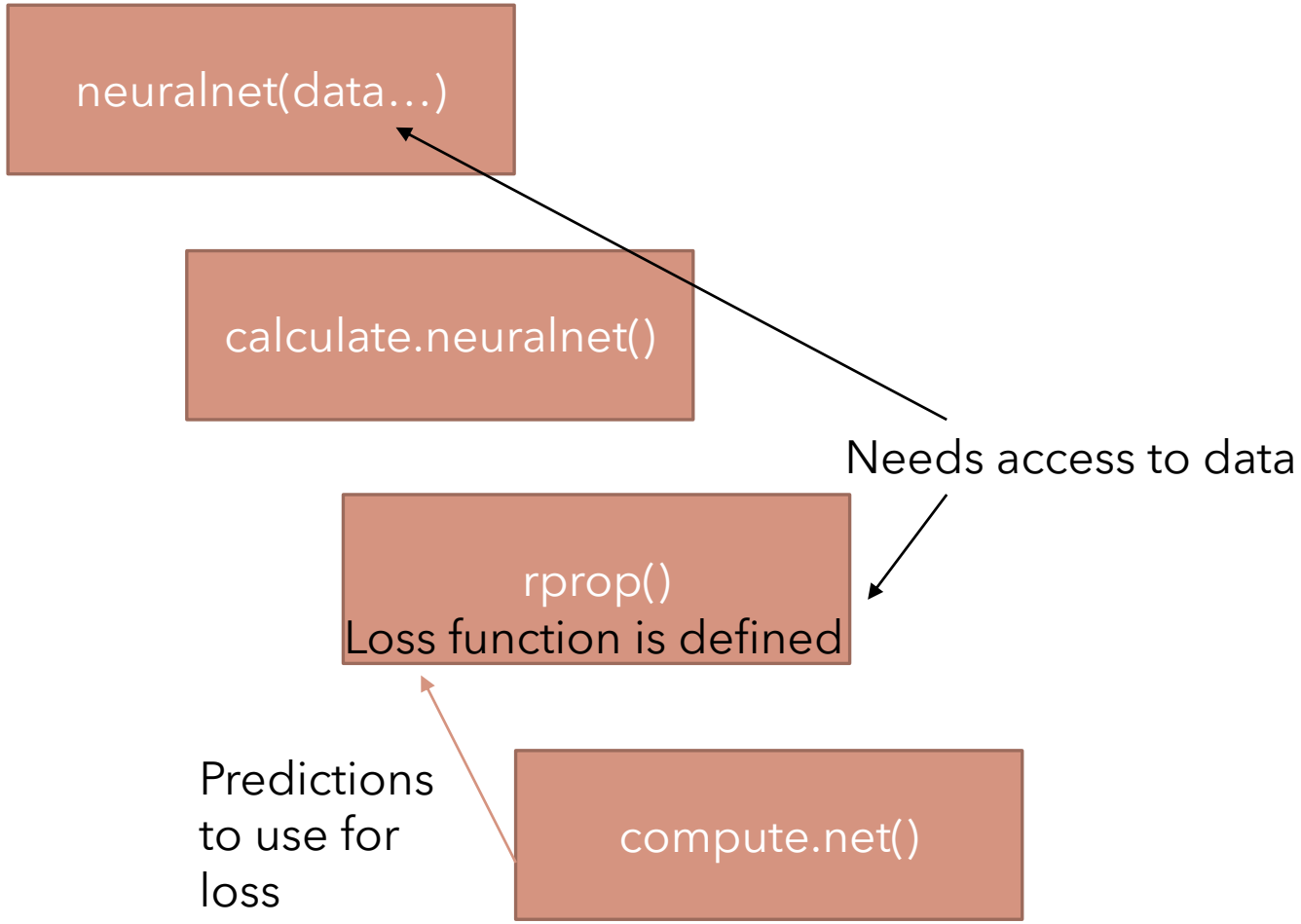
Loss function

- We could create a loss function that penalizes missing hard to predict or similar but different sample units
  - Would have to identify where in the code the loss is calculated and how
  - Would have to provide access to the training data within a loss function
  - Write the function
  - Would have to go through the repo to make all other code compatible with new loss function

```
rprop <-
  function (weights, response, covariate, threshold, learningrate.limit,
            learningrate.factor, stepmax, lifesign, lifesign.step, act.fct,
            act.deriv.fct, output.act.fct, output.act.deriv.fct, err.fct, err.deriv.fct, algorithm, linear.output,
            exclude, learningrate.bp)
  {
    step <- 1
    nchar.stepmax <- max(nchar(stepmax), 7)
    length.weights <- length(weights)
    nrow.weights <- sapply(weights, nrow)
    ncol.weights <- sapply(weights, ncol)
    length.unlist <- length(unlist(weights)) - length(exclude)
    learningrate <- as.vector(matrix(0.1, nrow = 1, ncol = length.unlist))
    gradients.old <- as.vector(matrix(0, nrow = 1, ncol = length.unlist))
    if (is.null(exclude))
      exclude <- length(unlist(weights)) + 1
    if (attr(act.fct, "type") == "tanh" || attr(act.fct, "type") == "logistic" || attr(act.fct, "type") == "relu")
      special <- TRUE
    else special <- FALSE
    if (attr(output.act.fct, "type") == "tanh" || attr(output.act.fct, "type") == "logistic" || attr(output.act.fct, "type") == "relu")
      output.special <- TRUE
    else output.special <- FALSE
    if (linear.output) {
      output.act.fct <- function(x) {
        x
      }
      output.act.deriv.fct <- function(x) {
        matrix(1, nrow(x), ncol(x))
      }
    }
    else {
      if (attr(err.fct, "type") == "ce" && attr(act.fct, "type") == "logistic") {
        err.deriv.fct <- function(x, y) {
          x * (1 - y) - y * (1 - x)
        }
        linear.output <- TRUE
      }
      #output.act.fct <- act.fct
      #output.act.deriv.fct <- act.deriv.fct
    }
    result <- compute.net(weights, length.weights, covariate = covariate,
                          act.fct = act.fct, act.deriv.fct = act.deriv.fct, output.act.fct = output.act.fct,
                          output.act.deriv.fct = output.act.deriv.fct, special, output.special)
    err.deriv <- err.deriv.fct(result$net.result, response)
    gradients <- calculate.gradients(weights = weights, length.weights = length.weights,
```

Missing “data”

Changing loss function: Cran package



How Cran NN Loss Function *Seems* to Work

There is a bunch of other stuff not on the diagram

Work-around?

Instead of another loss function, how would doubling the number of Hard to Predict and Similar but Different sample units compare?

Hard to Predict

```
Best Threshold: 0.25
Accuracy: 0.7611940298507462
Recall: 0.4144385026737968
Confusion Matrix:
[[916 117]
 [219 155]]

True Positive: 155
True Negative: 916
False Positive: 117
False Negative: 219

Accuracy: 0.7611940298507462
True-Positve Rate: 0.4144385026737968
```

Similar but Different

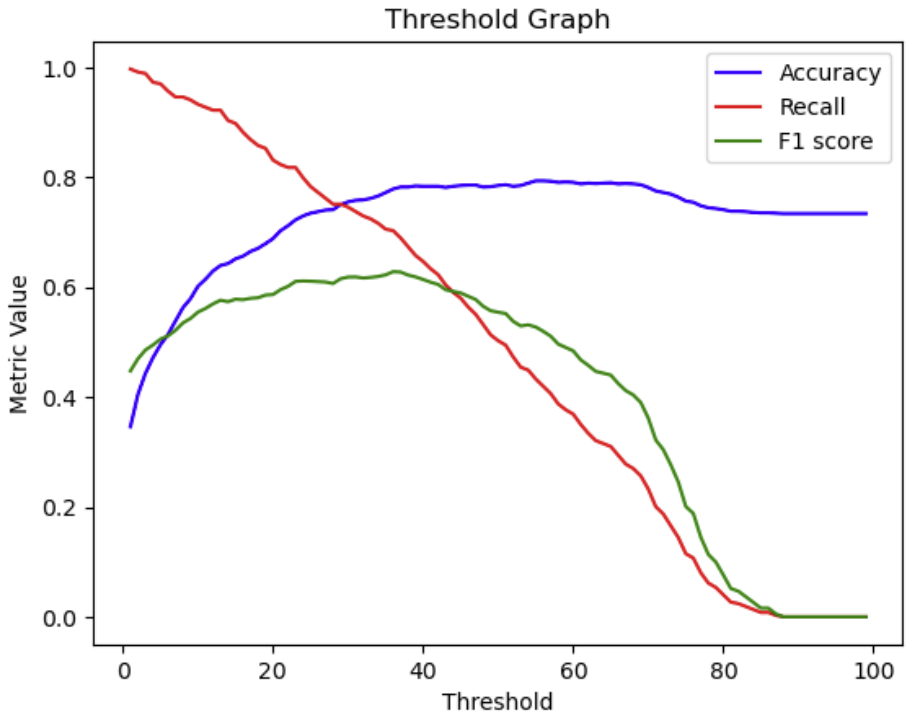
```
Best Threshold: 0.22
Accuracy: 0.7292110874200426
Recall: 0.7433155080213903
Confusion Matrix:
[[748 285]
 [ 96 278]]

True Positive: 278
True Negative: 748
False Positive: 285
False Negative: 96

Accuracy: 0.7292110874200426
True-Positve Rate: 0.7433155080213903
```

Week 3

Goals

1. Threshold log model
2. Better metric
3. Reorder hard to predict
4. Hard to predict threshold
5. Duplicate Hard to predict
6. Ordering saves time? (converges faster) (graph both on same graph)
7. Cluster (graphs)
8. Decision tree
    1. Methods
9. Harder to classify … why?
10. How to combine 2 neural networks

1. Threshold log model




```
Best Threshold: 0.36
Accuracy: 0.7789623312011372
Recall: 0.7032085561497327
F1: 0.6284348864994026
```

What This changes:
- "misses" dataset
  - Oversample misses
  - Hybrid oversample
- Probability threshold for determining "Hard to predict"

What this does not affect:
- Regular dataset
- Regular oversample
- Similar but different

2. Adding F1 score

Threshold is determined by F1 score

F1 score is logged after each epoch

```
def thresh(pred, ytest):
    bestacc = 0
    besttp = 0
    bestf1 = 0
    bestthresh = 0
    accList = []
    tpList = []
    f1list = []
    threshlist = []
    for i in range(1,100):
        classPred = [0 if val < (i/100) else 1 for val in pred]
        accuracy = accuracy_score(ytest, classPred)
        tpr = recall_score(ytest, classPred)
        f1 = f1_score(ytest, classPred)
        accList.append(accuracy)
        tpList.append(tpr)
        f1list.append(f1)
        threshlist.append(i)
        if (f1>bestf1):
            bestacc= accuracy
            besttp = tpr
            bestf1 = f1
            bestthresh=(i/100)

    plt.plot(threshlist, accList, 'b', label='Accuracy')
    plt.plot(threshlist, tpList, 'r', label='Recall')
    plt.plot(threshlist, f1list, 'g', label='F1 score')
    plt.title('Threshold Graph')
    plt.xlabel('Threshold')
    plt.ylabel('Metric Value')
    plt.legend()
    plt.show()
    print("Best Threshold:", bestthresh)
    print("Accuracy:", bestacc)
    print("Recall:", besttp)
    print("F1:", bestf1)

    return bestthresh
```

Threshold & F1 Effects

Number of (training) misses
- 1073 -> 1203

Oversample Misses
- 6698 -> 6828

Hybrid oversample
- Different missed sample units used

*note: test threshold = .36 | train threshold = .33
i.e. test set is not an outlier

```
Confusion Matrix:
[[917 116]
 [186 188]]

True Positive: 188
True Negative: 917
False Positive: 116
False Negative: 186

Accuracy: 0.7853589196872779
True-Positve Rate: 0.5026737967914439
F1 score: 0.5545722713864307
```

```
Confusion Matrix:
[[833 200]
 [111 263]]

True Positive: 263
True Negative: 833
False Positive: 200
False Negative: 111

Accuracy: 0.7789623312011372
True-Positve Rate: 0.7032085561497327
F1 score: 0.6284348864994026
```

3. Reorder Hard to Predict

```
with warnings.catch_warnings():
    warnings.simplefilter(action='ignore', category=FutureWarning)
    notHTP = pd.DataFrame().append(notC0Hp).append(notC1LP)
    HTP = pd.DataFrame()

    alternate = min(len(C0HP), len(C1LP))
    for i in range (0, alternate):
        HTP = HTP.append(C0HP.iloc[i]).append(C1LP.iloc[i])
        #print(len(orderedHTP))

    if (alternate == len(C0HP)):
        HTP = HTP.append(C1LP.iloc[(len(C0HP)):(len(C1LP))])

    else:
        HTP = HTP.append(C0HP.iloc[(len(C1LP)):(len(C0HP))])
```

	Churn_Yes	Prob	Class
935	0.0	0.836990	1.0
268	1.0	0.004228	0.0
4150	0.0	0.826079	1.0
4819	1.0	0.005233	0.0
293	0.0	0.823613	1.0

4. Hard to Predict Threshold

"Hard to Predict" can be defined as 1/2 std from the threshold.

*note: did not use distance from mean probability since mean = 0.2657157437464206 and std= 0.2464009321533121 3

```
def hardToPredict(std, threshold):

    upperlimit = threshold + std
    lowerlimit = threshold - std

    C0HP = class0[(class0["Prob"]>upperlimit)].sort_values(by="Prob", ascending = False)
    notC0Hp = class0[(class0["Prob"]<=upperlimit)]

    C1LP = class1[(class1["Prob"]<lowerlimit)].sort_values(by="Prob")
    notC1LP = class1[(class1["Prob"]>=lowerlimit)]

    with warnings.catch_warnings():
        warnings.simplefilter(action='ignore', category=FutureWarning)
        notHTP = pd.DataFrame().append(notC0Hp).append(notC1LP)
        HTP = pd.DataFrame()

        alternate = min(len(C0HP), len(C1LP))
        for i in range (0, alternate):
            HTP = HTP.append(C0HP.iloc[i]).append(C1LP.iloc[i])
            #print(len(orderedHTP))

        if (alternate == len(C0HP)):
            HTP = HTP.append(C1LP.iloc[(len(C0HP)):(len(C1LP))])

        else:
            HTP = HTP.append(C0HP.iloc[(len(C1LP)):(len(C0HP))])

    print("HTP:", len(HTP))
    print("notHTP:", len(notHTP))

    return (HTP, notHTP)
```

Updated Results

Regular

```
Best Threshold: 0.32
Accuracy: 0.775408670931059
Recall: 0.6898395721925134
F1: 0.6201923076923077
Confusion Matrix:
[[833 200]
 [116 258]]

True Positive: 258
True Negative: 833
False Positive: 200
False Negative: 116

Accuracy: 0.775408670931059
True-Positve Rate: 0.6898395721925134
F1 score: 0.6201923076923077
```

Oversample

```
Best Threshold: 0.9
Accuracy: 0.7121535181236673
Recall: 0.7272727272727273
F1: 0.5732349841938883
Confusion Matrix:
[[730 303]
 [102 272]]

True Positive: 272
True Negative: 730
False Positive: 303
False Negative: 102

Accuracy: 0.7121535181236673
True-Positve Rate: 0.7272727272727273
F1 score: 0.5732349841938883
```

Oversample Misses

```
Best Threshold: 0.3
Accuracy: 0.7761194029850746
Recall: 0.4197860962566845
F1: 0.4992050874403815
Confusion Matrix:
[[935  98]
 [217 157]]

True Positive: 157
True Negative: 935
False Positive: 98
False Negative: 217

Accuracy: 0.7761194029850746
True-Positve Rate: 0.4197860962566845
F1 score: 0.4992050874403815
```

Hybrid

```
Best Threshold: 0.6
Accuracy: 0.7725657427149965
Recall: 0.42245989304812837
F1: 0.49685534591194963
Confusion Matrix:
[[929 104]
 [216 158]]

True Positive: 158
True Negative: 929
False Positive: 104
False Negative: 216

Accuracy: 0.7725657427149965
True-Positve Rate: 0.42245989304812837
F1 score: 0.49685534591194963
```

Updated Results

Similar but different

```
Best Threshold: 0.16
Accuracy: 0.7540867093105899
Recall: 0.7540106951871658
F1: 0.6197802197802198
Confusion Matrix:
[[779 254]
 [ 92 282]]

True Positive: 282
True Negative: 779
False Positive: 254
False Negative: 92

Accuracy: 0.7540867093105899
True-Positve Rate: 0.7540106951871658
F1 score: 0.6197802197802198
```

Hard to predict

```
Best Threshold: 0.35
Accuracy: 0.7626154939587776
Recall: 0.7299465240641712
F1: 0.6204545454545455
Confusion Matrix:
[[800 233]
 [101 273]]

True Positive: 273
True Negative: 800
False Positive: 233
False Negative: 101

Accuracy: 0.7626154939587776
True-Positve Rate: 0.7299465240641712
F1 score: 0.6204545454545455
```

5. More Hard to predict

Single

```
Best Threshold: 0.35
Accuracy: 0.7626154939587776
Recall: 0.7299465240641712
F1: 0.6204545454545455
Confusion Matrix:
[[800 233]
 [101 273]]

True Positive: 273
True Negative: 800
False Positive: 233
False Negative: 101

Accuracy: 0.7626154939587776
True-Positve Rate: 0.7299465240641712
F1 score: 0.6204545454545455
```

Double

```
Best Threshold: 0.29
Accuracy: 0.46766169154228854
Recall: 0.9572192513368984
F1: 0.48873720136518767
Confusion Matrix:
[[300 733]
 [ 16 358]]

True Positive: 358
True Negative: 300
False Positive: 733
False Negative: 16

Accuracy: 0.46766169154228854
True-Positve Rate: 0.9572192513368984
F1 score: 0.48873720136518767
```

Triple

```
Best Threshold: 0.3
Accuracy: 0.6282871357498223
Recall: 0.8903743315508021
F1: 0.5601345668629099
Confusion Matrix:
[[551 482]
 [ 41 333]]

True Positive: 333
True Negative: 551
False Positive: 482
False Negative: 41

Accuracy: 0.6282871357498223
True-Positve Rate: 0.8903743315508021
F1 score: 0.5601345668629099
```

Quadruple

```
Best Threshold: 0.28
Accuracy: 0.496090973702914
Recall: 0.9411764705882353
F1: 0.4982307147912243
Confusion Matrix:
[[346 687]
 [ 22 352]]

True Positive: 352
True Negative: 346
False Positive: 687
False Negative: 22

Accuracy: 0.496090973702914
True-Positve Rate: 0.9411764705882353
F1 score: 0.4982307147912243
```

Note* HTP contain about double the amount of class 0 compared to class 1

Conclusions

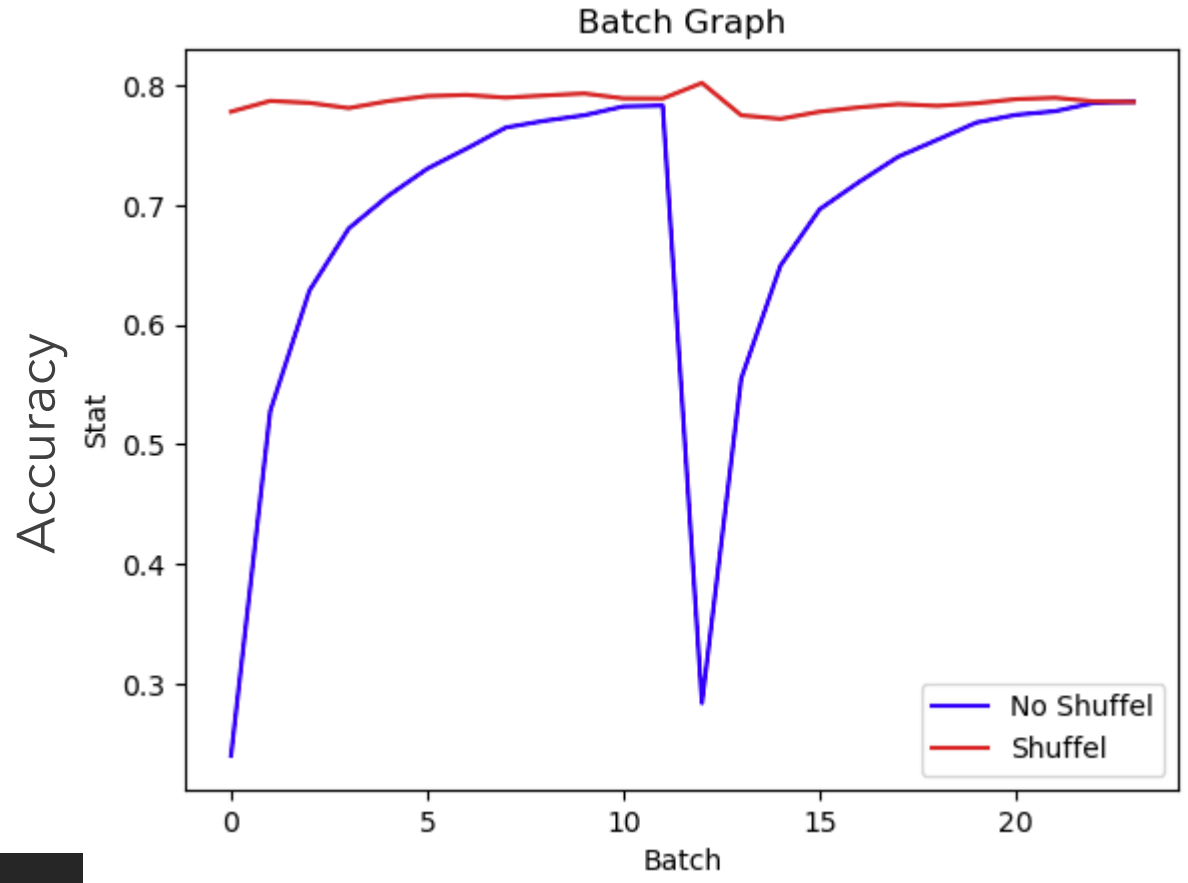
F1 score based thresholds increased accuracy on class 0 sample units

Note: Batch graphs (and epoch) with F1 were added after the completion of this slide (did not have time to update)

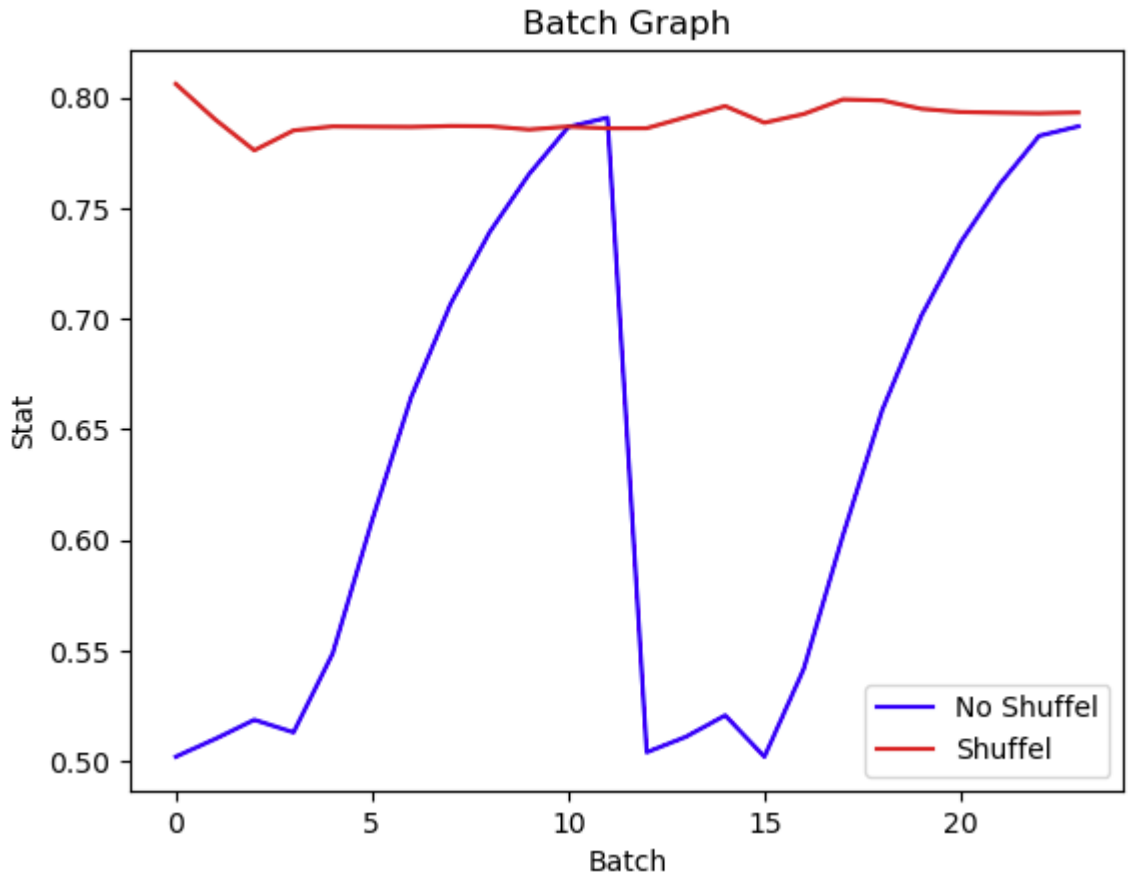
Also, considered adding a graph of the change in the metric (differenced graph)
Also, probably should have used more batches to show model progression

6. Batch graphs

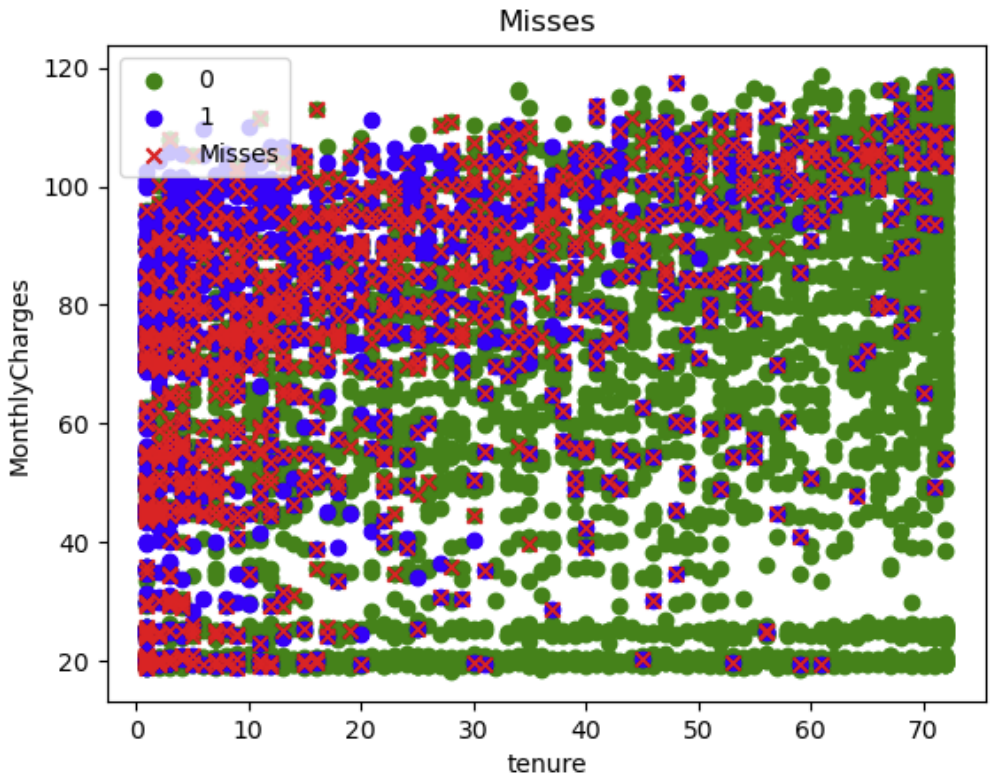
Hard to predict

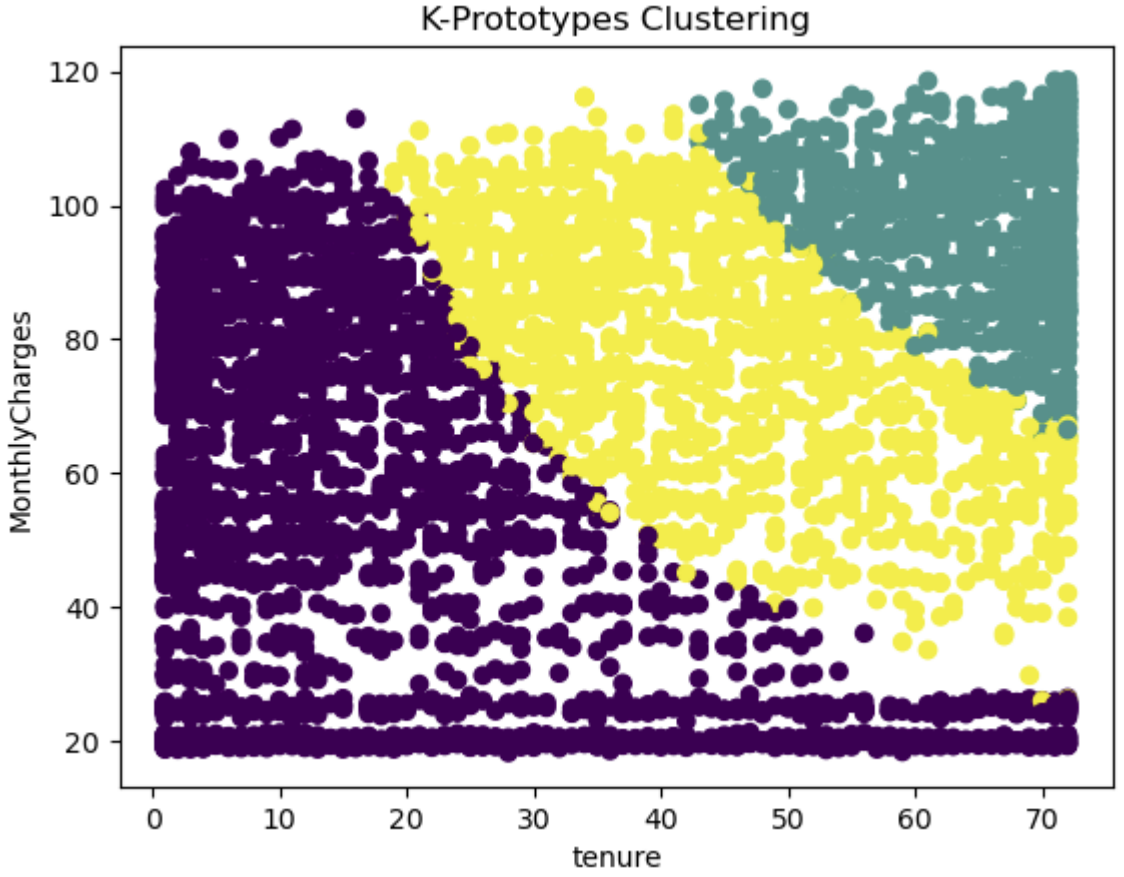


Similar but different

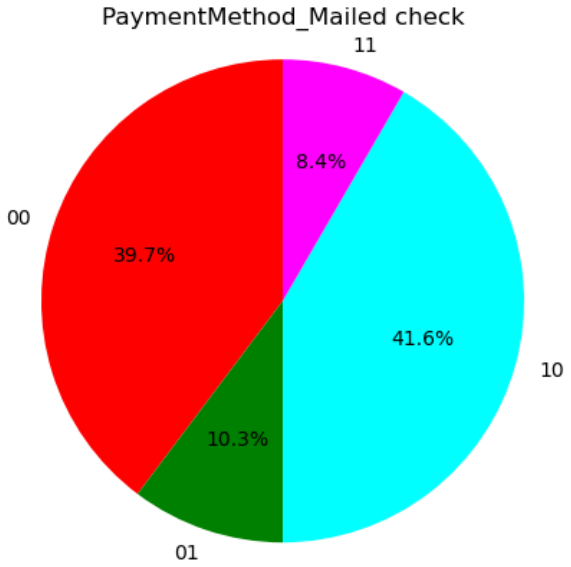


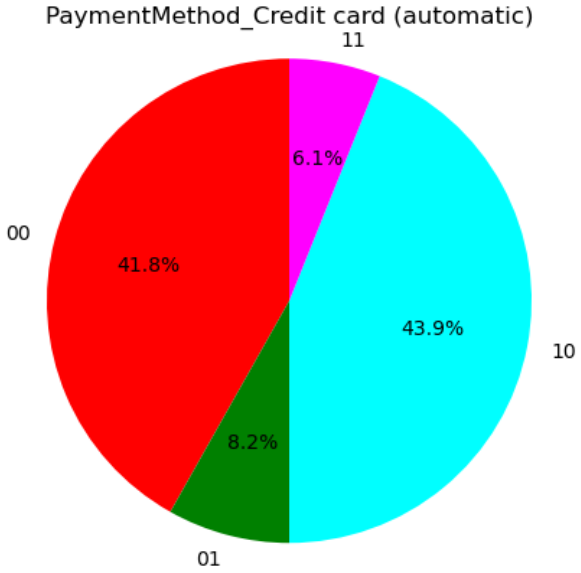
7. Cluster

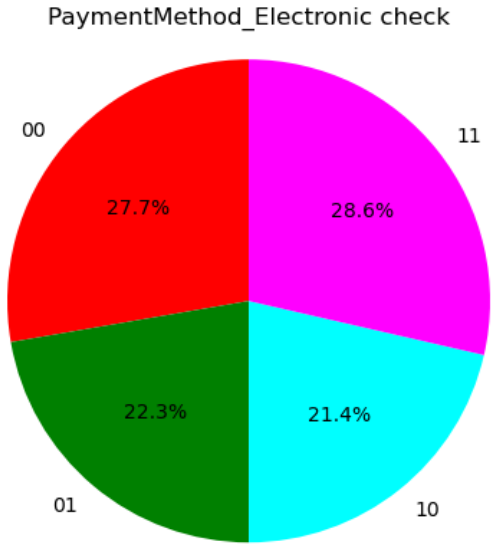

Misses
0
1
Misses
MonthlyCharges
tenure
120
100
80
60
40
20
0
10
20
30
40
50
60
70

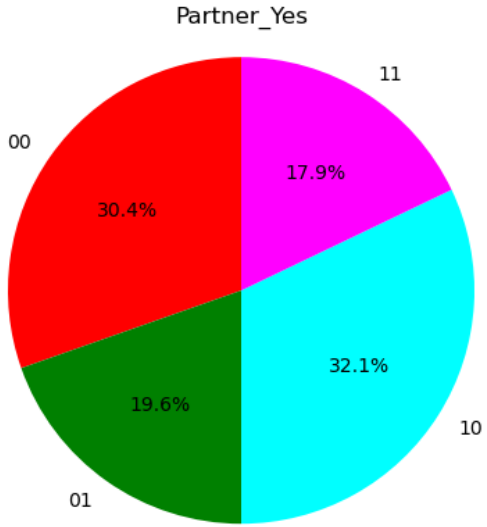

K-Prototypes Clustering
MonthlyCharges
tenure
120
100
80
60
40
20
0
10
20
30
40
50
60
70

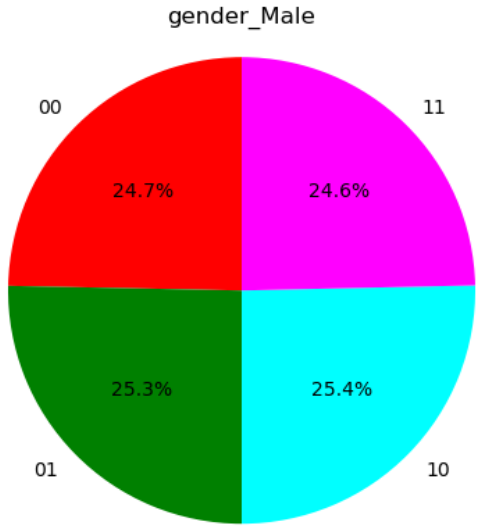
9. Graphs of Similar but different vars

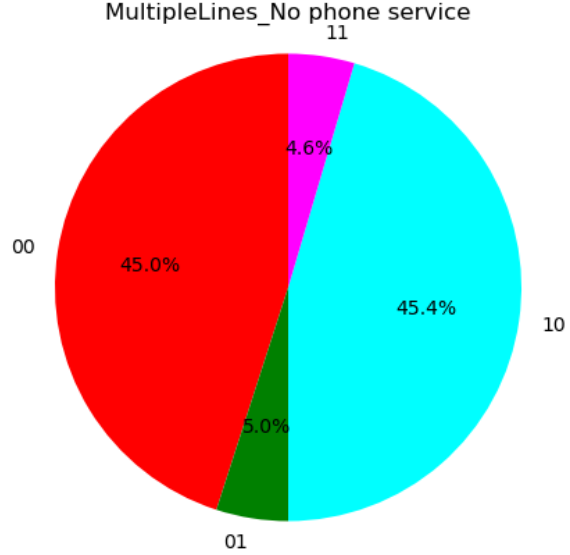


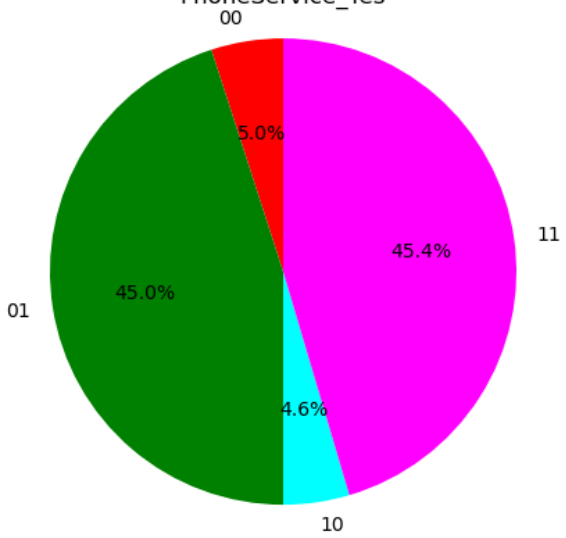


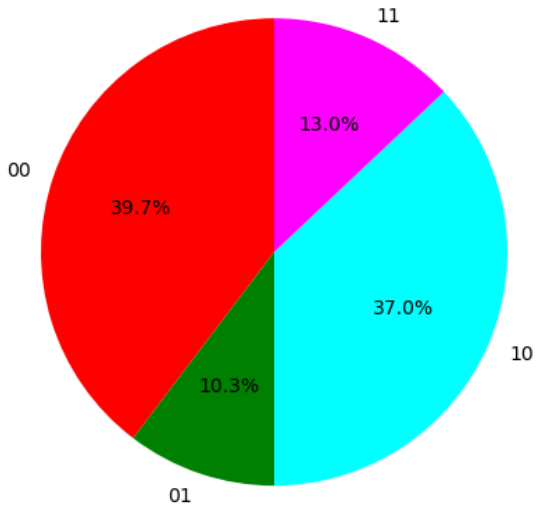


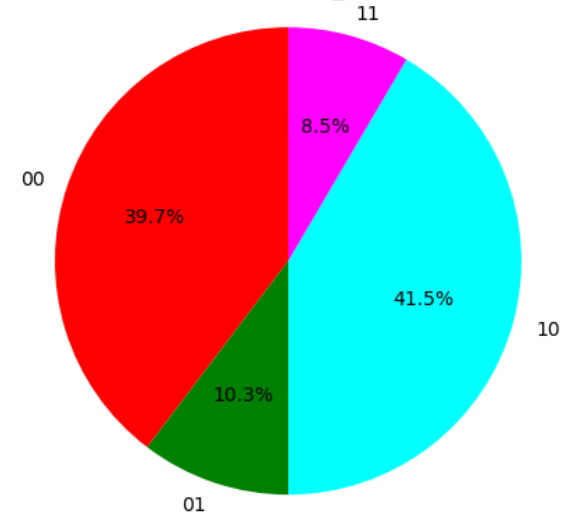

Partner_Yes
11
00
17.9%
30.4%
32.1%
19.6%
10
01

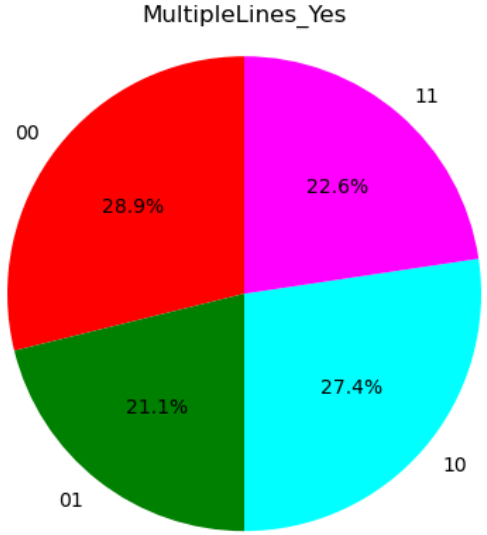

gender_Male
00
11
24.7%
24.6%
25.3%
25.4%
01
10

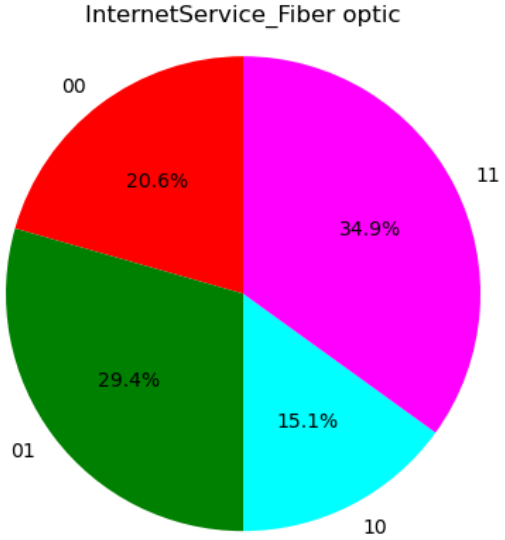

MultipleLines_No phone service
11
4.6%
00
45.0%
45.4%
10
5.0%
01

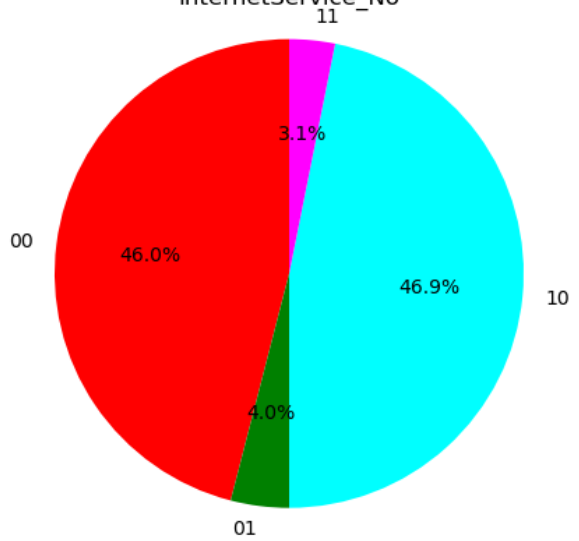

PhoneService_Yes
00
5.0%
11
45.4%
01
45.0%
4.6%
10

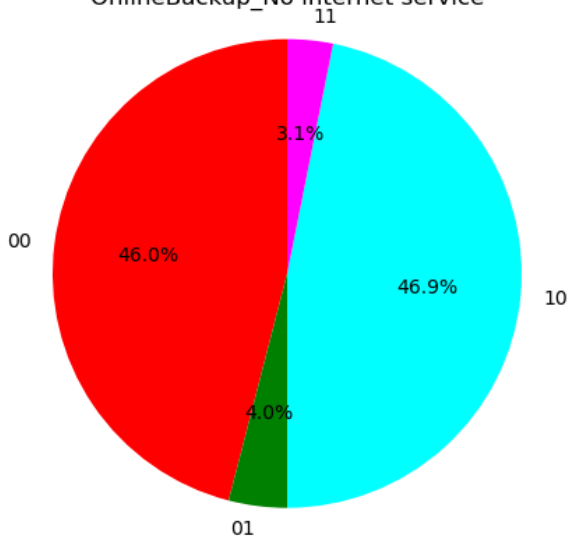

SeniorCitizen_1
11
13.0%
00
39.7%
37.0%
10
10.3%
01

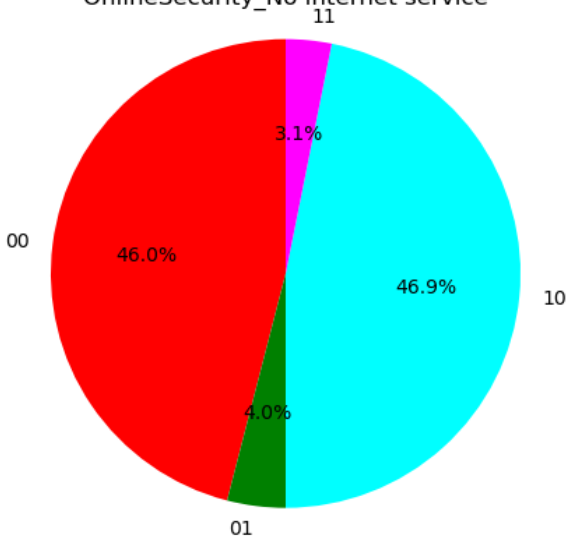

Dependents_Yes
11
8.5%
00
39.7%
41.5%
10
10.3%
01

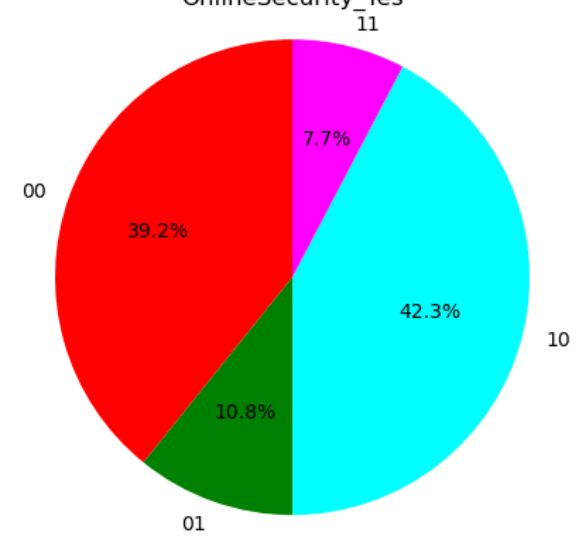

MultipleLines_Yes
11
00
22.6%
28.9%
21.1%
27.4%
10
01

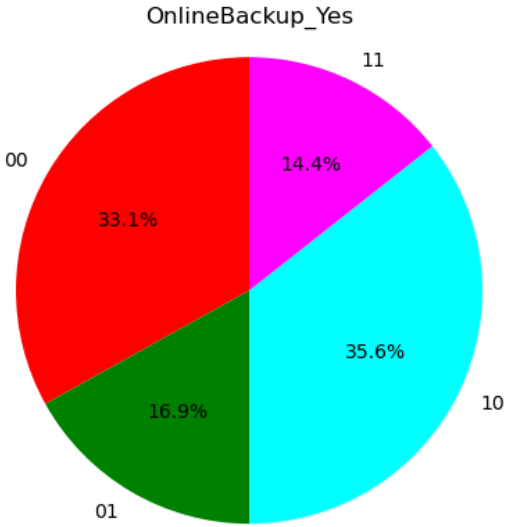

InternetService_Fiber optic
00
11
20.6%
34.9%
29.4%
15.1%
01
10

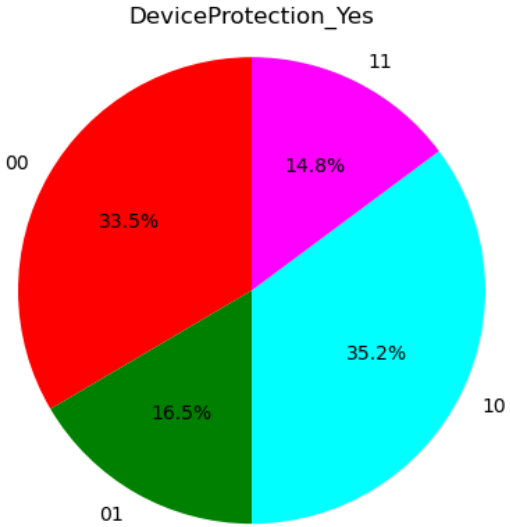

InternetService_No
11
3.1%
00
46.0%
46.9%
10
4.0%
01

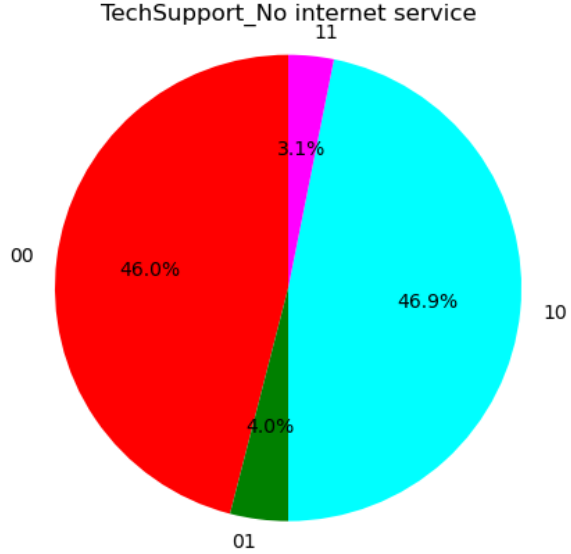

OnlineBackup_No internet service
11
3.1%
00
46.0%
46.9%
10
4.0%
01

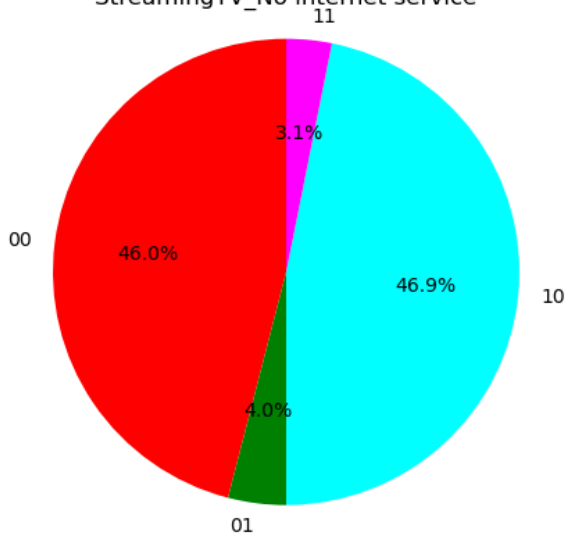

OnlineSecurity_No internet service
11
3.1%
00
46.0%
46.9%
10
4.0%
01

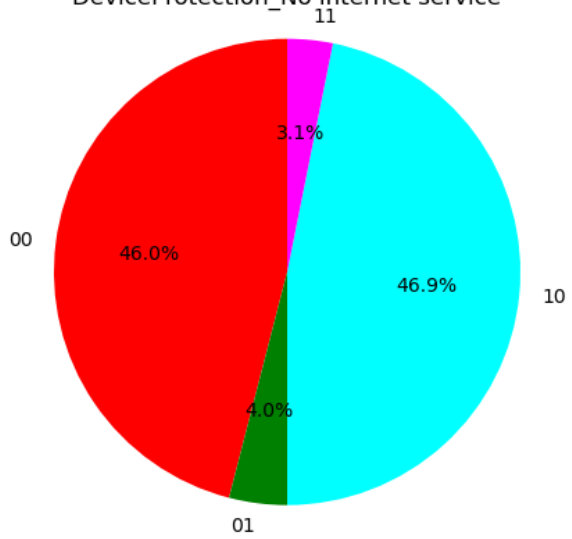

OnlineSecurity_Yes
11
7.7%
00
39.2%
42.3%
10
10.8%
01

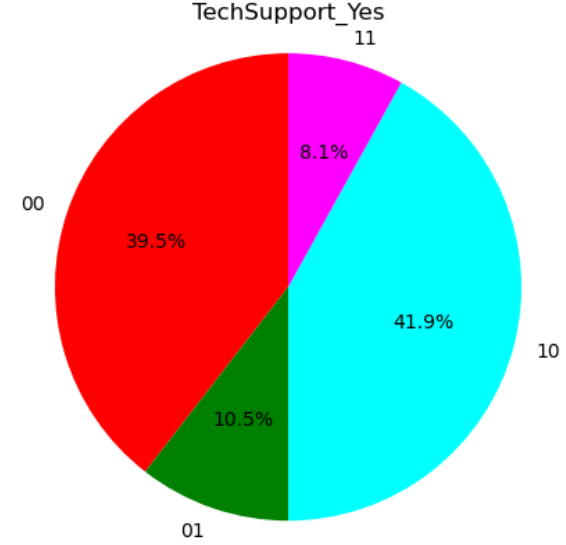

OnlineBackup_Yes
11
00
14.4%
33.1%
35.6%
10
16.9%
01

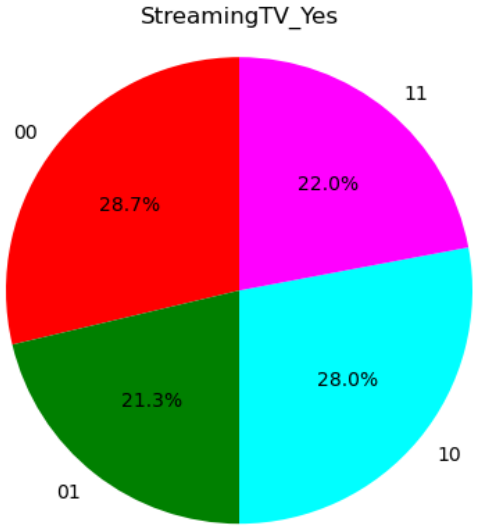

DeviceProtection_Yes
11
00
14.8%
33.5%
35.2%
10
16.5%
01

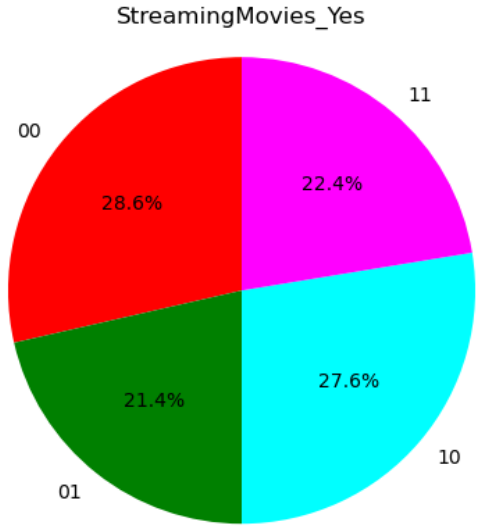

TechSupport_No internet service
11
3.1%
00
46.0%
46.9%
10
4.0%
01

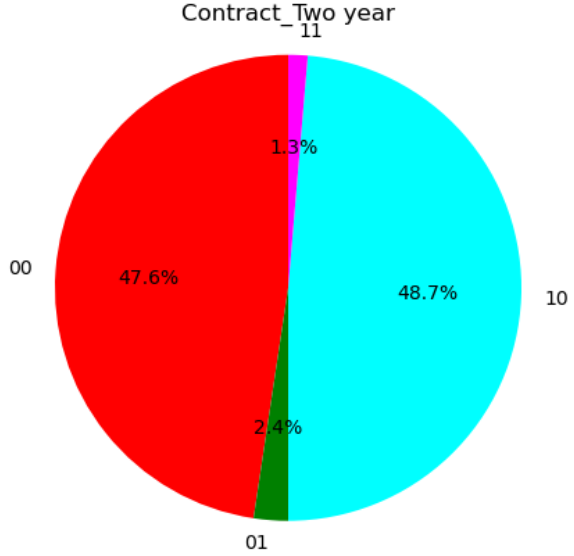

StreamingTV_No internet service
11
3.1%
00
46.0%
46.9%
10
4.0%
01

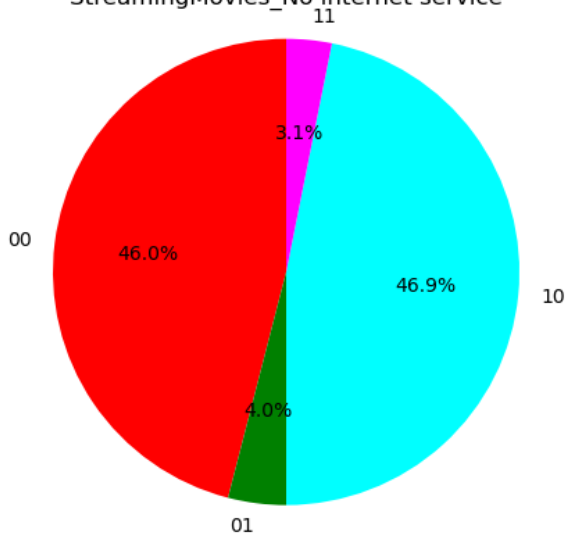

DeviceProtection_No internet service
11
3.1%
00
46.0%
46.9%
10
4.0%
01

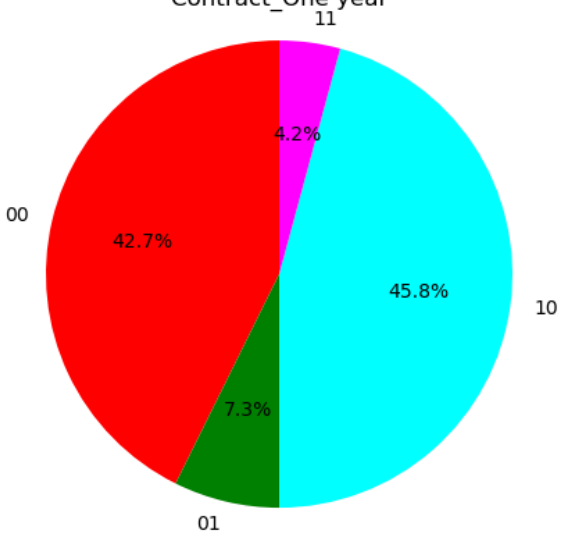

TechSupport_Yes
11
8.1%
00
39.5%
41.9%
10
10.5%
01

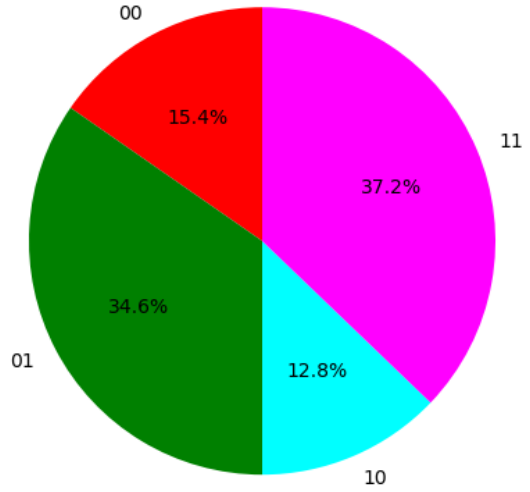

StreamingTV_Yes
11
00
22.0%
28.7%
21.3%
28.0%
10
01


StreamingMovies_Yes
11
00
22.4%
28.6%
21.4%
27.6%
10
01


Contract_Two year
11
1.3%
00
47.6%
48.7%
10
2.4%
01


StreamingMovies_No internet service
11
3.1%
00
46.0%
46.9%
10
4.0%
01


Contract_One year
11
4.2%
00
42.7%
45.8%
10
7.3%
01

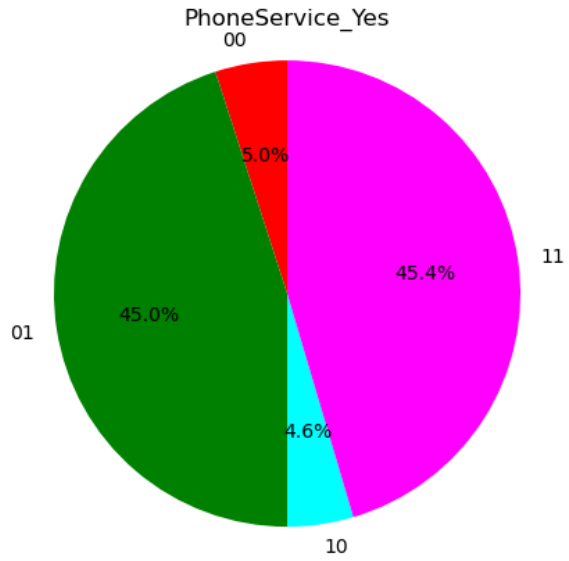

PaperlessBilling_Yes
00
15.4%
11
37.2%
34.6%
01
12.8%
10

Conclusion

The graphs show that, for the similar but different sample units, the reason they are hard to distinguish is because none of the categorical variables give a clear indication as to which class the unit belongs to

For example: If a SBD customer has phone service they are about equally likely to be in class 1 or class 0

This is similar for all SBD variables

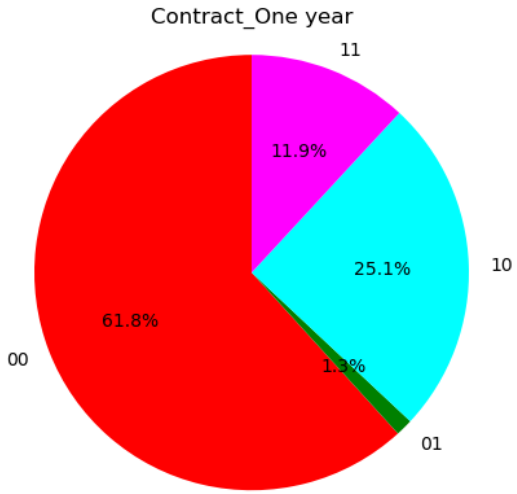


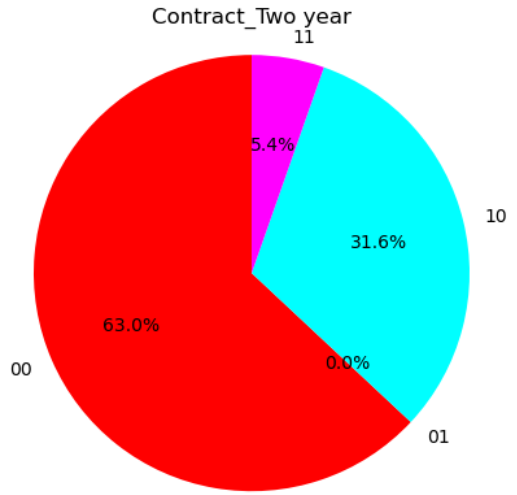
Graphs of Hard to Predict

In many of the hard to predict variable graphs, variables seem to be more useful.

Ex: If a HTP sample unit has a one-year contract, then they are much more likely to be in class 1 than class 0

If a HTP sample unit has a two-year contract, then they are in class 1





Conclusion

If we perform preprocessing to determine hard to predict sample units, a specialized (HTP) neural network should theoretically be able to take advantage of the significant variables to make predictions on otherwise hard to predict units

To get a better visualization of which variables are important, we can use trees

Decision Tree

Specifics: used all variables and did not specify max depth, learning rate = 0.1

Did not evaluate tree effectiveness due to time limitations (planning on doing this)

Did not print tree due to size

Full Dataset Tree

```
--------------------Regular Tree--------------------
                              Importance
tenure                          0.211222
MonthlyCharges                  0.191139
TotalCharges                    0.188157
InternetService_Fiber optic     0.111978
gender_Male                     0.029432
--------------------Random Forest--------------------
                                Importance
TotalCharges                      0.192555
tenure                            0.172762
MonthlyCharges                    0.171334
InternetService_Fiber optic       0.038679
PaymentMethod_Electronic check    0.037415
--------------------Gradient Boost--------------------
                                Importance
tenure                            0.312747
InternetService_Fiber optic       0.207994
PaymentMethod_Electronic check    0.093715
TotalCharges                      0.076709
MonthlyCharges                    0.063578
```

Tree: SBD

```
--------------------Regular Tree--------------------
                 Importance
MonthlyCharges     0.260927
TotalCharges       0.231734
tenure             0.123790
Partner_Yes        0.036721
gender_Male        0.036453
--------------------Random Forest--------------------
                 Importance
TotalCharges       0.211438
MonthlyCharges     0.203258
tenure             0.157152
gender_Male        0.036066
Partner_Yes        0.030687
--------------------Gradient Boost--------------------
                             Importance
tenure                         0.279648
TotalCharges                   0.225330
MonthlyCharges                 0.215213
InternetService_Fiber optic    0.046790
Contract_One year              0.037441
```

Tree: HTP

```
--------------------Regular Tree--------------------
                                    Importance
tenure                                0.324358
OnlineSecurity_No internet service    0.277141
OnlineSecurity_Yes                    0.088383
InternetService_Fiber optic           0.079887
MonthlyCharges                        0.051443
--------------------Random Forest--------------------
                              Importance
tenure                          0.228381
TotalCharges                    0.117853
InternetService_Fiber optic     0.077295
MonthlyCharges                  0.073391
OnlineSecurity_Yes              0.069306
--------------------Gradient Boost--------------------
                                     Importance
tenure                                 0.369566
InternetService_Fiber optic            0.172143
StreamingMovies_No internet service    0.066579
OnlineSecurity_Yes                     0.060917
InternetService_No                     0.058853
```

Conclusion

Seems like categorical variables are not very useful for the SBD units compared to the HTP units

Combine 2 NN

- Separate Hard to predict and not

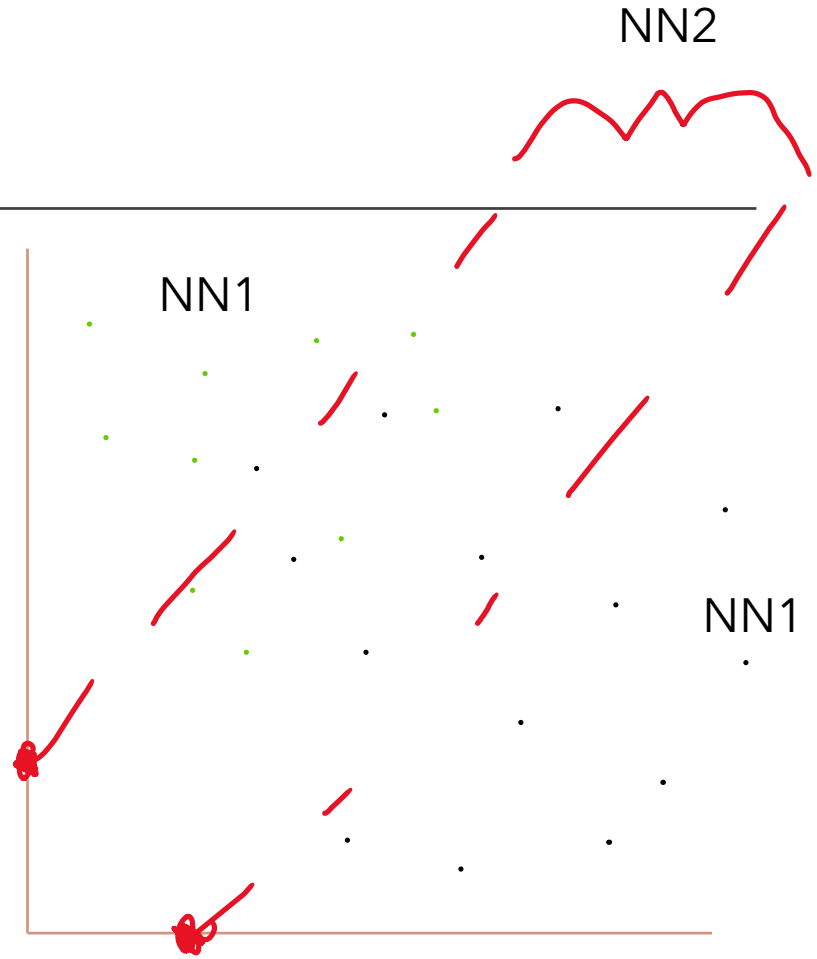
- Train 2 models

- Using test data, determine if the sample units are closer in distance to HTP or regular

- get prediction from respective NN

- similar to support vector idea
  - NN1 handles sample units that are similar to data points away from the support vectors
  - NN2 handles sample units similar to sample units within support vector range
  - Maybe we use SVM instead of Log model?

Similar to time series data: Use NN to find pattern in “non-stationary” data



Week 4

To-Do List

1. Weighted probability sampling
2. Boundary sample units (possibly save)
3. Randomness in the test set or NN (run multiple times w/ diff test and train) (bootstrap test)
4. Validation set accuracy
5. Calculate tree metrics
6. Boosting with 1 var each tree
7. 2 NN

1. Weighted probably sampling

Method 1: order the training data so that is consists of n batches with m units. Each batch will be sampled using weighted probabilities. Then feed the data into a NN with shuffle = false, and make the batch size = m. This way, each batch is sampled with a weighted probably, and its faster to implement than editing source code

Downside: each epoch is still the same <- (working on it)

1. Weighted probably sampling

```python
def weightedBatches(higherWeightdf, lowerWeightdf, weightMultiplier, unitsPerBatch):
    numBatches = math.floor((len(higherWeightdf) + len(lowerWeightdf))/unitsPerBatch)

    high = higherWeightdf.copy()
    low = lowerWeightdf.copy()

    low["weight"] = 1
    high["weight"] = weightMultiplier

    joint = pd.concat([low, high])
    weighted = pd.DataFrame()

    numWeighted = []

    for i in range(numBatches):
        sample = joint.sample(n = unitsPerBatch, replace=False, weights='weight')
        joint = joint.drop(sample.index)

        numWeighted.append(len(sample[sample["weight"] == weightMultiplier]))
        weighted = pd.concat([weighted, sample])

    weighted = pd.concat([weighted, joint])

    mp = math.floor(len(numWeighted)/2)

    print("Weighted units per batch in first half:", sum(numWeighted[0:mp])/mp)

    print("Weighted units per batch in second half:", sum(numWeighted[mp:])/(len(numWeighted)-mp))

    return weighted.drop(["weight"], axis=1)
```

```python
weightedHTP = weightedBatches(HTP, notHTP, 3, 32)
```

```
Weighted units per batch in first half: 6.9655172413793105
Weighted units per batch in second half: 1.1363636363636365
```

```
weightedHTP.head(5)
```

	tenure	MonthlyCharges	TotalCharges	gender_Male	SeniorCitizen_1	Partner_Yes	Dependents_Yes	PhoneS
1017	64.0	24.40	1601.20	0.0	0.0	0.0	0.0	
5638	1.0	19.95	19.95	1.0	0.0	0.0	0.0	
1997	67.0	105.65	6717.90	0.0	0.0	1.0	1.0	
1532	14.0	19.60	300.40	1.0	0.0	0.0	0.0	
1266	4.0	80.25	303.70	0.0	1.0	0.0	0.0	

5 rows × 33 columns

```
x = weightedHTP.drop(["Churn_Yes", "Prob", "Class"], axis = 1)
y = weightedHTP["Churn_Yes"]

class printeverybatch(tf.keras.Model):
    def train_step(self, data):
        x, y = data
        tf.print('new batch:')
        tf.print(x, summarize=-1)
        tf.print(y, summarize=-1)
        return super().train_step(data)

inputs=tf.keras.Input((30,))
model=printeverybatch(inputs, tf.keras.layers.Dense(1)(inputs))

model.compile(loss=tf.keras.losses.MeanSquaredError(), optimizer=tf.keras.optimizers.SGD())

model.fit(x,y,batch_size=5, verbose=2, epochs=2, shuffle=False)
```

```
Epoch 1/2
new batch:
[[64 24.4 1601.2 0 0 0 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0]
 [1 19.95 19.95 1 0 0 0 1 0 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 1 0]
 [67 105.65 6717.9 0 0 1 1 1 0 1 1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 1 0 0 0]
 [14 19.6 300.4 1 0 0 0 1 0 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 1 0 0]
 [4 80.25 303.7 0 1 0 0 1 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0]]
```

Proving that #1 works

2. Boundary sample units

Range: threshold +- .5*ST.D.

# of rows: 1104 (HTP~700)

```
upperlimit = threshold + stdProb
lowerlimit = threshold - stdProb
```

```
save = probdf[probdf['Prob'].between(lowerlimit, upperlimit)]
```

```
len(save)
```

```
1104
```

3. Randomness in the test set & NN

Same training and test set

*Correction from last week, the test and training set have stayed the same entirely

```
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size=0.2, random_state=42)
```

```
NN(probdf.drop(["Class", "Prob"], axis=1), xtest, ytest)
```

Test run	Test F1
1	0.6217494089834514
2	0.6310559006211179
3	0.6195899772209567
4	0.6248548199767712
5	0.6279357231149567

Interesting:

```
NN(probdf.drop(["Class", "Prob"], axis=1), xtest, ytest)
```


```
Best Threshold: 0.32
Accuracy: 0.7725657427149965
Recall: 0.7032085561497327
F1: 0.6217494089834514
Confusion Matrix:
[[824 209]
 [111 263]]

True Positive: 263
True Negative: 824
False Positive: 209
False Negative: 111

Accuracy: 0.7725657427149965
True-Positve Rate: 0.7032085561497327
F1 score: 0.6217494089834514
```

```
#Log results
evaluate(ytest,ypred)
```

```
Confusion Matrix:
[[833 200]
 [111 263]]

True Positive: 263
True Negative: 833
False Positive: 200
False Negative: 111

Accuracy: 0.7789623312011372
True-Positve Rate: 0.7032085561497327
F1 score: 0.6284348864994026
```

3. Randomness in the test set & NN

Random training and test set

Test run	Test F1
1	0.6073697585768741
2	0.6347150259067358
3	0.6083086053412462
4	0.6274065685164212
5	0.6272321428571428
6	0.6409691629955947
7	0.6477272727272728
8	0.6513157894736842

```
test = xtest.copy()
test["Churn_Yes"] = ytest

BSF1 = []

for i in range(30):
    bootstrapTest = test.iloc[np.random.choice(len(test), size=len(test), replace=True)]
    xtest = bootstrapTest.drop('Churn_Yes', axis=1)
    ytest = bootstrapTest['Churn_Yes']

    pred = model.predict(xtest)
    bestThresh = thresh(pred, ytest)
    classPred = [0 if val < bestThresh else 1 for val in pred]
    f1 = f1_score(ytest, classPred)
    BSF1.append(f1)
```

```
print(max(BSF1))
```

```
0.6588541666666666
```

```
print(min(BSF1))
```

```
0.5855421686746988
```

```
print(sum(BSF1)/len(BSF1))
```

```
0.6187867657616579
```

3. Bootstrap Test set

Validation loss and metrics are evaluated at the end of each epoch, there is no per-batch validation loss and metrics as computed by Keras.

4. Validation set batch graph

Need custom call back

The API does not support, but still you can do it. Below is a Pseudo Code. If Time permits I shall try out and put a cleaner version.

```
def on_batch_end(self, batch, logs={}):
     if batch%10 == 0: # Better dont evaluate for all Batch
        x_val = self.validation_data[0]
        y_val_true = self.validation_data[1]
        y_val_pred = self.model.predict(x_val)
        loss = self.compiled_loss(y_val_true, y_val_pred,regularization_losses=self.
        self.history[str(batch)+'_Loss'].append(loss)
```

Share Improve this answer Follow edited Mar 20 at 14:30 community wiki 3 revs user2458922

Add a comment

You can use a custom call back like

```
class MultipleValidationCallBack(keras.callbacks.Callback):

    def on_batch_end(self, epoch, logs=None):
        for thisValidationType in validationSets:
            #datagen =
            thisLoss = self.model.evaluate(validationSets[thisValidationType])
            logs[thisValidationType+"_loss"] = thisLoss[0]
            logs[thisValidationType+"_acc"] = thisLoss[1]
```

4. Validation set batch graph

validation_data was coming up “none”

```
print(history.validation_data)
```

```
None
```

Found the solution to this problem. Mention in issue comments https://github.com/keras-team/keras/issues/10472

```
class Metrics(Callback):

    def __init__(self, val_data, batch_size = 20):
        super().__init__()
        self.validation_data = val_data
        self.batch_size = batch_size
```

Initializing the validation data solves the problem when using data generators.

4. Validation set batch graph

Changed to .5
Should it calculate the best threshold?

```
#accuracy after each batch
class BCP(tf.keras.callbacks.Callback):
    batch_accuracy = [] # accuracy at given batch
    batch_f1 = [] # f1 at given batch
    batch_f1_val = [] # f1 of validation at given batch

    def __init__(self, val_data):
        super(BCP,self).__init__()
        self.validation_data = val_data

    def on_train_batch_end(self, batch, logs=None):
        x_val = self.validation_data[0]
        y_val_true = self.validation_data[1]
        y_val_pred = self.model.predict(x_val, verbose=0)

        y_val_class = [0 if val < .05 else 1 for val in y_val_pred]
        batchF1 = f1_m(y_val_true, y_val_class)

        BCP.batch_f1_val.append(batchF1)
        BCP.batch_accuracy.append(logs.get('accuracy'))
        BCP.batch_f1.append(logs.get('f1_m'))
```

Results combining everything

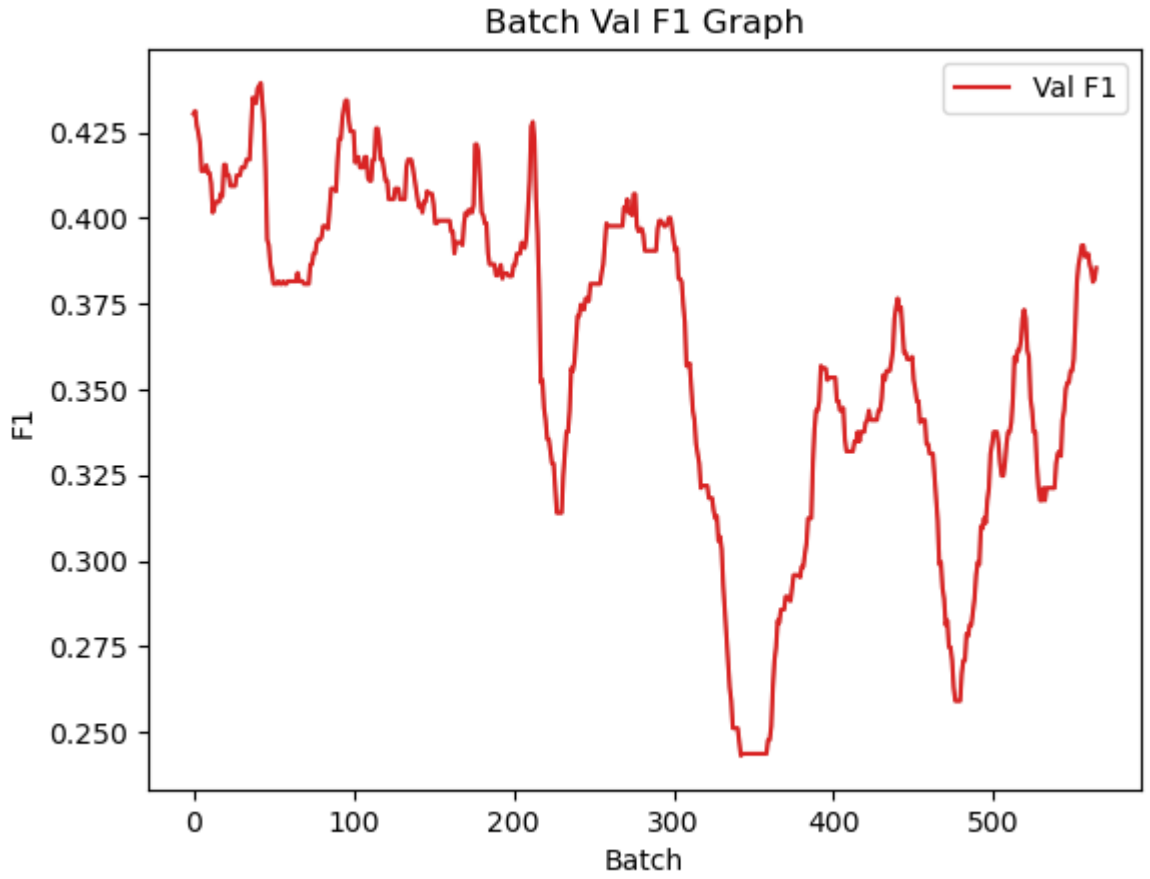
Using random weighted sampling (RWS), we can plot the batch graph showing the validation set f1 at the end of each batch. At the end, we can then bootstrap the test set to get the overall f1.

Using the following DFs: HTP (RWS), save (RWS), SBD (Haven't figured out how to RWS)

*Note: Training takes forever with added computation on every batch, so the number of epochs is drastically lower (200-> 5) and batch size is increased (32->50). ie. Expect worse performance and possibly higher variance

Regular(SH)

```
Best Threshold: 0.32
Accuracy: 0.775408670931059
Recall: 0.6898395721925134
F1: 0.6201923076923077
Confusion Matrix:
[[833 200]
 [116 258]]

True Positive: 258
True Negative: 833
False Positive: 200
False Negative: 116

Accuracy: 0.775408670931059
True-Positve Rate: 0.6898395721925134
F1 score: 0.6201923076923077
```

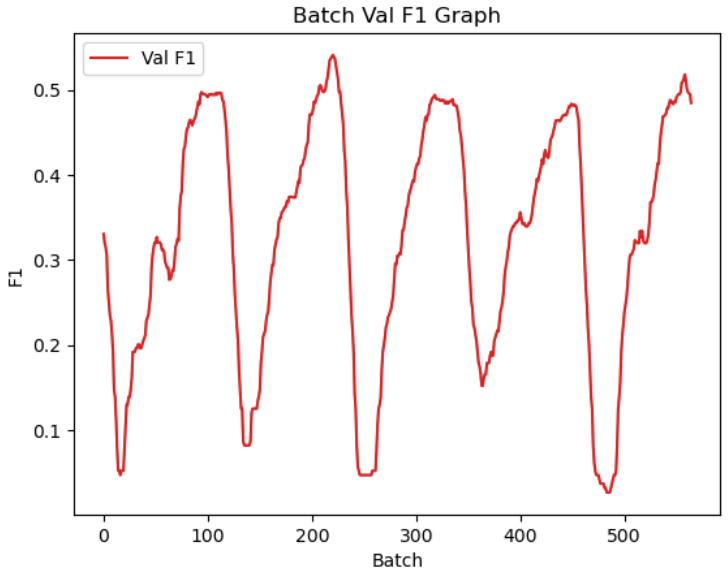



```
Best Threshold: 0.24
Accuracy: 0.7334754797441365
Recall: 0.7807486631016043
F1: 0.6089676746611054
Confusion Matrix:
[[740 293]
 [ 82 292]]

True Positive: 292
True Negative: 740
False Positive: 293
False Negative: 82
```

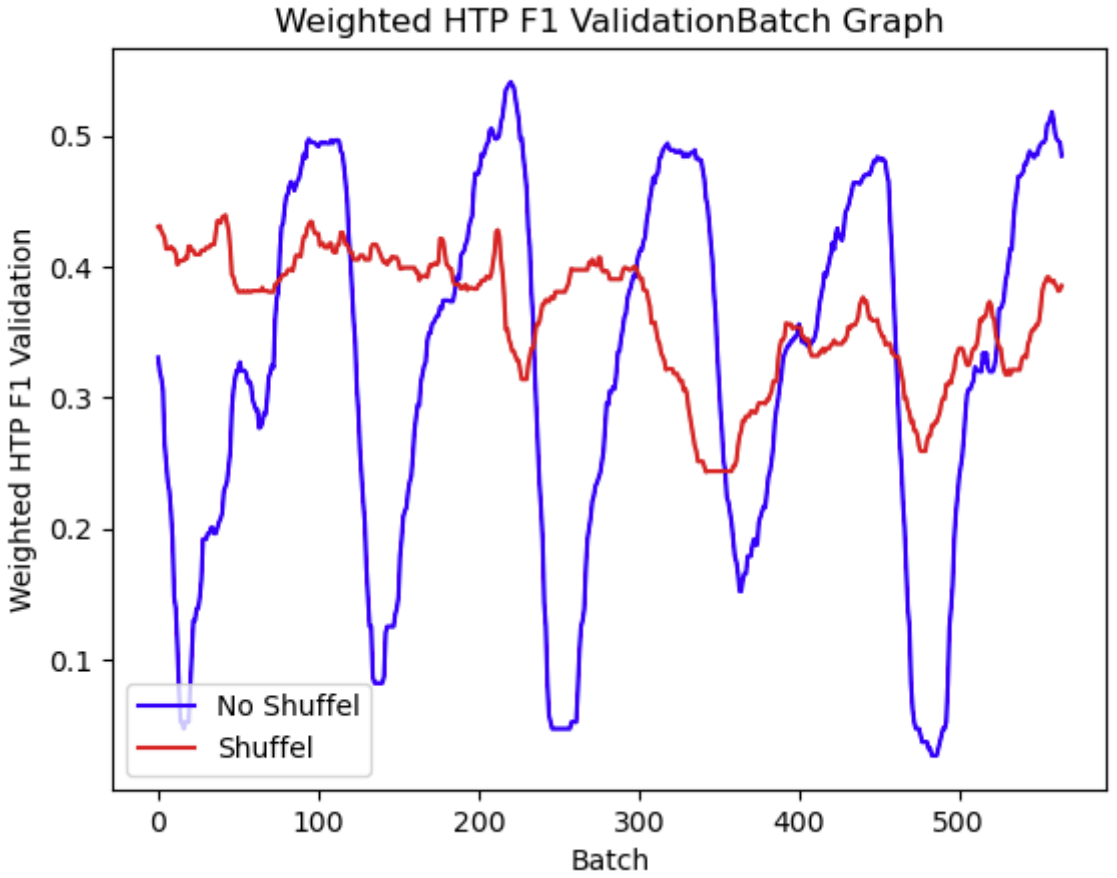
Bootstrapped: 0.6151338702764233

HTP(RWS)

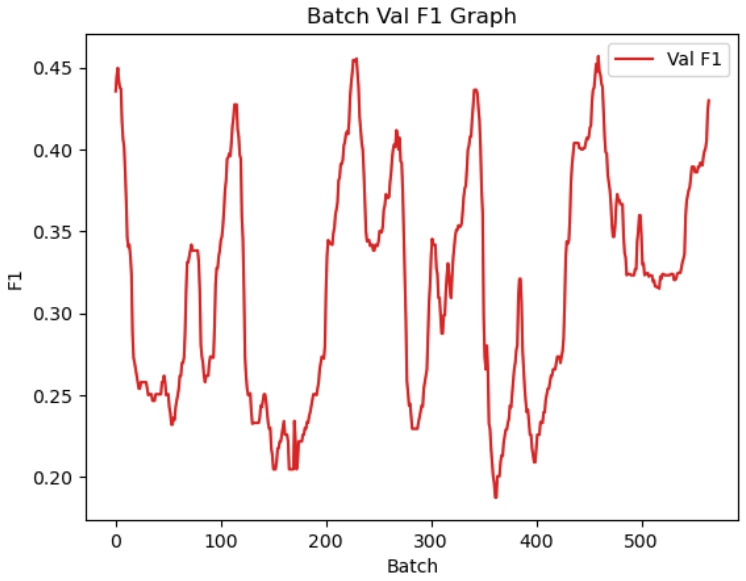
```
Best Threshold: 0.27
Accuracy: 0.7619047619047619
Recall: 0.679144385026738
F1: 0.6026097271648873
Confusion Matrix:
[[818 215]
 [120 254]]

True Positive: 254
True Negative: 818
False Positive: 215
False Negative: 120
```



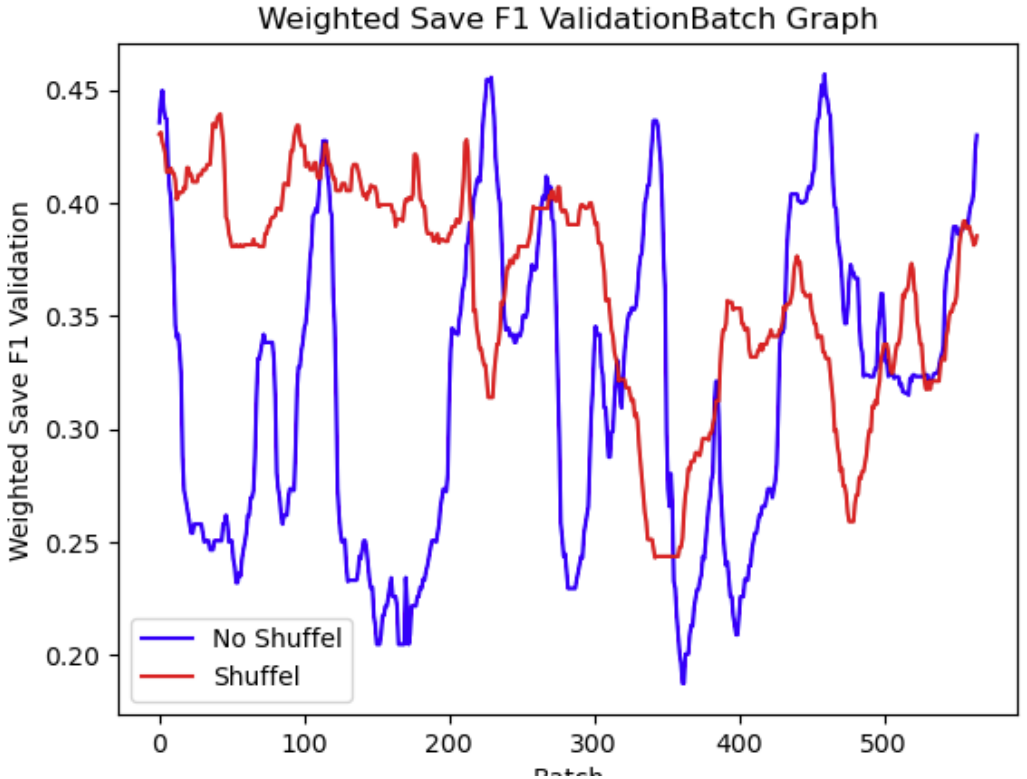

```
Best Threshold: 0.21
Accuracy: 0.7540867093105899
Recall: 0.7058823529411765
F1: 0.6041189931350114
Confusion Matrix:
[[797 236]
 [110 264]]

True Positive: 264
True Negative: 797
False Positive: 236
False Negative: 110

Accuracy: 0.7540867093105899
True-Positve Rate: 0.7058823529411765
F1 score: 0.6041189931350114
Bootstrapped: 0.605710068055737
```



Save(RWS)

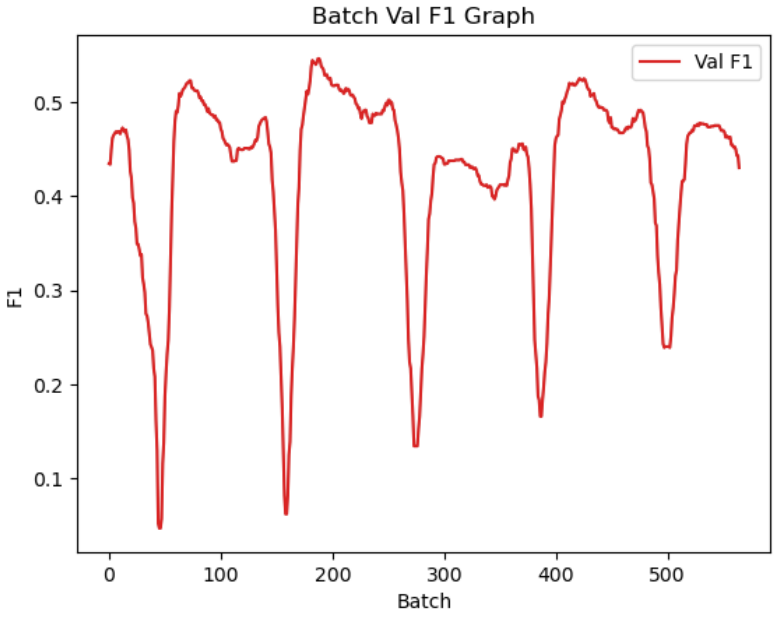



```
Best Threshold: 0.21
Accuracy: 0.7654584221748401
Recall: 0.6978609625668449
F1: 0.6126760563380282
Confusion Matrix:
[[816 217]
 [113 261]]

True Positive: 261
True Negative: 816
False Positive: 217
False Negative: 113

Accuracy: 0.7654584221748401
True-Positve Rate: 0.6978609625668449
F1 score: 0.6126760563380282
Bootstrapped: 0.6172493315578895
```

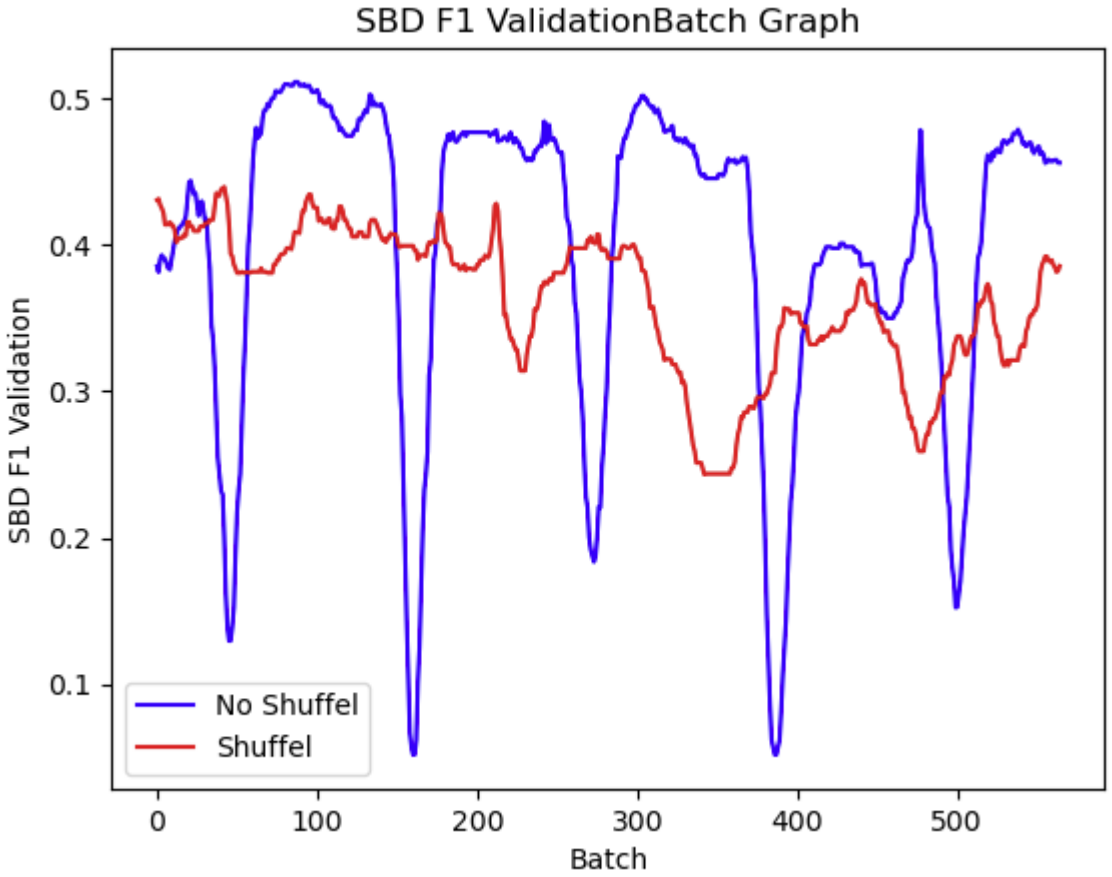


SBD

```
Best Threshold: 0.2
Accuracy: 0.7540867093105899
Recall: 0.5775401069518716
F1: 0.5552699228791773
Confusion Matrix:
[[845 188]
 [158 216]]

True Positive: 216
True Negative: 845
False Positive: 188
False Negative: 158
```




```
Best Threshold: 0.14
Accuracy: 0.7448471926083866
Recall: 0.5828877005347594
F1: 0.5484276729559747
Confusion Matrix:
[[830 203]
 [156 218]]

True Positive: 218
True Negative: 830
False Positive: 203
False Negative: 156

Accuracy: 0.7448471926083866
True-Positve Rate: 0.5828877005347
F1 score: 0.5484276729559747
Bootstrapped: 0.54916732177508
```



```
def dtree(x, y, xtest, ytest):
    dtc = DecisionTreeClassifier()
    rfc = RandomForestClassifier(n_estimators=100, random_state=42)
    gbc = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, random_state=42, max_depth = 1)

    dtc.fit(x,y)
    rfc.fit(x,y)
    gbc.fit(x,y)

    print("--------------------Regular Tree--------------------")
    dtcimp = varImp(dtc, x)
    print(dtcimp.sort_values("Importance", ascending= False).head(5))
    yfit = dtc.predict(x)
    print("Training:")
    evaluate(y, yfit)
    ypred = dtc.predict(xtest)
    print("Testomg:")
    evaluate(ytest, ypred)


    print("--------------------Random Forest--------------------")
    rfcimp = varImp(rfc, x)
    print(rfcimp.sort_values("Importance", ascending= False).head(5))
    yfit = rfc.predict(x)
    ypred = rfc.predict(xtest)
    print("Training:")
    evaluate(y, yfit)
    print("Testing:")
    evaluate(ytest, ypred)

    print("--------------------Gradient Boost--------------------")
    gbcimp = varImp(gbc, x)
    print(gbcimp.sort_values("Importance", ascending= False).head(5))
    yfit = gbc.predict(x)
    ypred = gbc.predict(xtest)
    print("Training:")
    evaluate(y, yfit)
    print("Testing:")
    evaluate(ytest, ypred)
```

5&6. Tree Metrics/GBM

Tree results: Full Dataset

```
--------------------Regular Tree--------------------
                              Importance
tenure                          0.215261
MonthlyCharges                  0.193188
TotalCharges                    0.191281
InternetService_Fiber optic     0.111978
gender_Male                     0.027858
Training:
Confusion Matrix:
[[4128    2]
 [  11 1484]]

True Positive: 1484
True Negative: 4128
False Positive: 2
False Negative: 11

Accuracy: 0.9976888888888888
True-Positve Rate: 0.9926421404682274
F1 score: 0.9956390472995639
Testomg:
Confusion Matrix:
[[820 213]
 [180 194]]

True Positive: 194
True Negative: 820
False Positive: 213
False Negative: 180

Accuracy: 0.720682302771855
True-Positve Rate: 0.5187165775401069
F1 score: 0.4967989756722151
```

```
--------------------Random Forest--------------------
                              Importance
TotalCharges                    0.192555
tenure                          0.172762
MonthlyCharges                  0.171334
InternetService_Fiber optic     0.038679
PaymentMethod_Electronic check  0.037415
Training:
Confusion Matrix:
[[4123    7]
 [   6 1489]]

True Positive: 1489
True Negative: 4123
False Positive: 7
False Negative: 6

Accuracy: 0.9976888888888888
True-Positve Rate: 0.9959866220735786
F1 score: 0.9956536275493147
Testing:
Confusion Matrix:
[[929 104]
 [195 179]]

True Positive: 179
True Negative: 929
False Positive: 104
False Negative: 195

Accuracy: 0.7874911158493249
True-Positve Rate: 0.4786096256684492
F1 score: 0.5449010654490107
```

```
--------------------Gradient Boost--------------------
                              Importance
tenure                          0.395961
InternetService_Fiber optic     0.255896
Contract_Two year               0.110512
PaymentMethod_Electronic check  0.098308
Contract_One year               0.037365
Training:
Confusion Matrix:
[[3843  287]
 [ 816  679]]

True Positive: 679
True Negative: 3843
False Positive: 287
False Negative: 816

Accuracy: 0.8039111111111111
True-Positve Rate: 0.45418060200668897
F1 score: 0.55180820804551
Testing:
Confusion Matrix:
[[960  73]
 [225 149]]

True Positive: 149
True Negative: 960
False Positive: 73
False Negative: 225

Accuracy: 0.7882018479033405
True-Positve Rate: 0.3983957219251337
F1 score: 0.5
```

Tree results: Just SBD

```
--------------------Regular Tree----------------------------------------Random Forest----------------------------------------Gradient Boost--------------------
                  Importance                                                Importance                                                Importance
MonthlyCharges      0.271924                                TotalCharges      0.211438                tenure                                0.460359
TotalCharges        0.235041                                MonthlyCharges    0.203258                InternetService_Fiber optic           0.202630
tenure              0.115322                                tenure            0.157152                PaymentMethod_Electronic check        0.121074
Partner_Yes         0.036137                                gender_Male       0.036066                TotalCharges                          0.080641
gender_Male         0.035707                                Partner_Yes       0.030687                MonthlyCharges                        0.033583
Training:                                                   Training:                                 Training:
Confusion Matrix:                                           Confusion Matrix:                         Confusion Matrix:
[[1493    2]                                                [[1488    7]                              [[1025  470]
 [  11 1484]]                                                [   6 1489]]                             [ 609  886]]

True Positive: 1484                                         True Positive: 1489                       True Positive: 886
True Negative: 1493                                         True Negative: 1488                       True Negative: 1025
False Positive: 2                                           False Positive: 7                         False Positive: 470
False Negative: 11                                          False Negative: 6                         False Negative: 609

Accuracy: 0.9956521739130435                                Accuracy: 0.9956521739130435              Accuracy: 0.6391304347826087
True-Positve Rate: 0.9926421404682274                       True-Positve Rate: 0.9959866220735786     True-Positve Rate: 0.5926421404682274
F1 score: 0.9956390472995639                                F1 score: 0.9956536275493147              F1 score: 0.6215363030515609
Testomg:                                                    Testing:                                  Testing:
Confusion Matrix:                                           Confusion Matrix:                         Confusion Matrix:
[[617 416]                                                  [[701 332]                                [[895 138]
 [159 215]]                                                  [180 194]]                               [163 211]]

True Positive: 215                                          True Positive: 194                        True Positive: 211
True Negative: 617                                          True Negative: 701                        True Negative: 895
False Positive: 416                                         False Positive: 332                       False Positive: 138
False Negative: 159                                         False Negative: 180                       False Negative: 163

Accuracy: 0.5913290689410092                                Accuracy: 0.6361051883439943             Accuracy: 0.7860696517412935
True-Positve Rate: 0.5748663101604278                       True-Positve Rate: 0.5187165775401069     True-Positve Rate: 0.5641711229946524
F1 score: 0.4278606965174129                                F1 score: 0.43111111111111106            F1 score: 0.5836791147994468
```

Numerical variables

Able to separate sample units with similar probabilities but different classes really well (overfit)

Imp cat. vars. are intuitively irrelevant

High Test F1
Just SBD trained can predict whole test set well

Tree results: Just HTP

```
--------------------Regular Tree--------------------
                                  Importance
tenure                              0.324358
DeviceProtection_No internet service    0.277141
OnlineSecurity_Yes                  0.088383
MonthlyCharges                      0.071280
InternetService_Fiber optic         0.068887
Training:
Confusion Matrix:
[[445   0]
 [  0 261]]

True Positive: 261
True Negative: 445
False Positive: 0
False Negative: 0

Accuracy: 1.0
True-Positve Rate: 1.0
F1 score: 1.0
Testomg:
Confusion Matrix:
[[237 796]
 [264 110]]

True Positive: 110
True Negative: 237
False Positive: 796
False Negative: 264

Accuracy: 0.24662402274342574
True-Positve Rate: 0.29411764705882354
F1 score: 0.171875
```

```
--------------------Random Forest--------------------
                                  Importance
tenure                              0.228381
TotalCharges                        0.117853
InternetService_Fiber optic         0.077295
MonthlyCharges                      0.073391
OnlineSecurity_Yes                  0.069306
Training:
Confusion Matrix:
[[445   0]
 [  0 261]]

True Positive: 261
True Negative: 445
False Positive: 0
False Negative: 0

Accuracy: 1.0
True-Positve Rate: 1.0
F1 score: 1.0
Testing:
Confusion Matrix:
[[232 801]
 [269 105]]

True Positive: 105
True Negative: 232
False Positive: 801
False Negative: 269

Accuracy: 0.23951670220326937
True-Positve Rate: 0.2807486631016043
F1 score: 0.16406250000000003
```

```
--------------------Gradient Boost--------------------
                                  Importance
tenure                              0.387729
InternetService_Fiber optic         0.200589
OnlineSecurity_Yes                  0.083580
TechSupport_No internet service     0.062141
TechSupport_Yes                     0.053201
Training:
Confusion Matrix:
[[442   3]
 [  9 252]]

True Positive: 252
True Negative: 442
False Positive: 3
False Negative: 9

Accuracy: 0.9830028328611898
True-Positve Rate: 0.9655172413793104
F1 score: 0.9767441860465116
Testing:
Confusion Matrix:
[[251 782]
 [266 108]]

True Positive: 108
True Negative: 251
False Positive: 782
False Negative: 266

Accuracy: 0.25515280739161333
True-Positve Rate: 0.2887700534759358
F1 score: 0.17088607594936708
```

Tree results: Just save

```
--------------------Regular Tree----------------------------------------Random Forest--------------------
                                Importance                                    Importance
MonthlyCharges                    0.246576   TotalCharges           0.215215
TotalCharges                      0.229257   MonthlyCharges         0.195170
tenure                            0.128415   tenure                 0.148982
PaymentMethod_Electronic check    0.043619   gender_Male            0.036164
Partner_Yes                       0.041412   PaperlessBilling_Yes   0.031529
Training:                                    Training:
Confusion Matrix:                            Confusion Matrix:
[[728   0]                                   [[727   1]
 [  1 375]]                                   [  0 376]]

True Positive: 375                           True Positive: 376
True Negative: 728                           True Negative: 727
False Positive: 0                            False Positive: 1
False Negative: 1                            False Negative: 0

Accuracy: 0.9990942028985508                 Accuracy: 0.9990942028985508
True-Positve Rate: 0.9973404255319149        True-Positve Rate: 1.0
F1 score: 0.9986684420772304                 F1 score: 0.99867197875166
Testomg:                                     Testing:
Confusion Matrix:                            Confusion Matrix:
[[616 417]                                   [[923 110]
 [232 142]]                                   [288  86]]

True Positive: 142                           True Positive: 86
True Negative: 616                           True Negative: 923
False Positive: 417                          False Positive: 110
False Negative: 232                          False Negative: 288

Accuracy: 0.5387348969438521                 Accuracy: 0.7171286425017769
True-Positve Rate: 0.37967914438502676       True-Positve Rate: 0.22994652406417113
F1 score: 0.30439442658092175                F1 score: 0.3017543859649123
```

```
--------------------Gradient Boost--------------------
                                Importance
tenure                            0.433249
TotalCharges                      0.328231
MonthlyCharges                    0.148724
Partner_Yes                       0.033251
InternetService_Fiber optic       0.025587
Training:
Confusion Matrix:
[[717  11]
 [359  17]]

True Positive: 17
True Negative: 717
False Positive: 11
False Negative: 359

Accuracy: 0.6648550724637681
True-Positve Rate: 0.04521276595744681
F1 score: 0.08415841584158416
Testing:
Confusion Matrix:
[[1013   20]
 [ 364   10]]

True Positive: 10
True Negative: 1013
False Positive: 20
False Negative: 364

Accuracy: 0.7270788912579957
True-Positve Rate: 0.026737967914438502
F1 score: 0.0495049504950495
```

7. 2NN

3 ways to use specialized models for test sets

1. Find some way to determine which sample units bet fit specialized criteria (which ones are HTP) and get prediction from respective model
2. Use weights
3. Use a NN to determine if the sample unit is HTP

To test if option 1 or 3 works in practice, I will find which sample units are HTP in the test set as a check to see if feeding into specialized models actually works

In an effort to allow for more training of the NN, I will not be recording per-batch data

7. 2NN: Save

```
Best Threshold: 0.28
Accuracy: 0.7690120824449183
Recall: 0.7245989304812834
F1: 0.6251441753171857
Confusion Matrix:
[[811 222]
 [103 271]]

True Positive: 271
True Negative: 811
False Positive: 222
False Negative: 103
```

Save NN

```
Best Threshold: 0.01
Accuracy: 0.3736654804270463
Recall: 1.0
F1: 0.5440414507772021
Confusion Matrix:
[[  0 176]
 [  0 105]]

True Positive: 105
True Negative: 0
False Positive: 176
False Negative: 0

Accuracy: 0.3736654804270463
True-Positve Rate: 1.0
F1 score: 0.5440414507772021
Starting Model Training
Model Training Finished
Training ROC:
142/142 [==============================] - 0s 855us/step
```

Not Save NN

```
Best Threshold: 0.4
Accuracy: 0.8161634103019538
Recall: 0.7137546468401487
F1: 0.6497461928934011
Confusion Matrix:
[[727 130]
 [ 77 192]]

True Positive: 192
True Negative: 727
False Positive: 130
False Negative: 77

Accuracy: 0.8161634103019538
True-Positve Rate: 0.7137546468401487
F1 score: 0.6497461928934011
```

Total

```
Total Confusion Matrix:
[[727 306]
 [ 77 297]]

Accuracy: 0.7277896233120114
True-Positve Rate: 0.7941176470588235
F1 score: 0.607983623336745
```

7. 2NN: HTP

```
Best Threshold: 0.28
Accuracy: 0.7690120824449183
Recall: 0.7245989304812834
F1: 0.6251441753171857
Confusion Matrix:
[[811 222]
 [103 271]]

True Positive: 271
True Negative: 811
False Positive: 222
False Negative: 103
```

HTP NN

```
Best Threshold: 0.01
Accuracy: 1.0
Recall: 1.0
F1: 1.0
Confusion Matrix:
[[131   0]
 [  0  72]]

True Positive: 72
True Negative: 131
False Positive: 0
False Negative: 0
```

Not HTP NN

```
Best Threshold: 0.7
Accuracy: 0.9111295681063123
Recall: 0.7483443708609272
F1: 0.8085867620751342
Confusion Matrix:
[[871  31]
 [ 76 226]]

True Positive: 226
True Negative: 871
False Positive: 31
False Negative: 76
```

Total

```
Total Confusion Matrix:
[[1002   31]
 [  76  298]]

Accuracy: 0.923951670220327
True-Positve Rate: 0.7967914438502673
F1 score: 0.847795163584637 2
```

Conclusion

NN is really good at predicting and training on HTP and not HTP separately

Only problem is how to separate actual test set data into HTP and not HTP

- NN to predict HTP?

Maybe try trees?

Also, stopped here because this week was getting very long

Didn’t have time to finish trying trees

Week 5

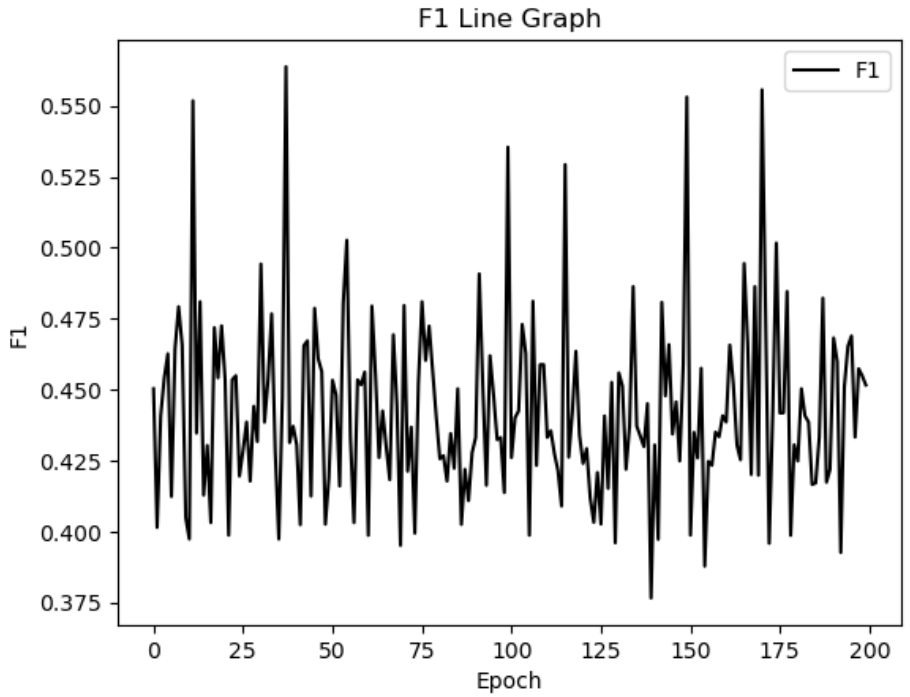
To-Do

1. Recompile model before fit
2. Weighted shuffle after each epoch
3. How to use for test set
   - See if one model has a pattern with how they deal with HTP
   - Weighted prediction
   - HTP Tree
4. Different dataset for separating (imbalanced)

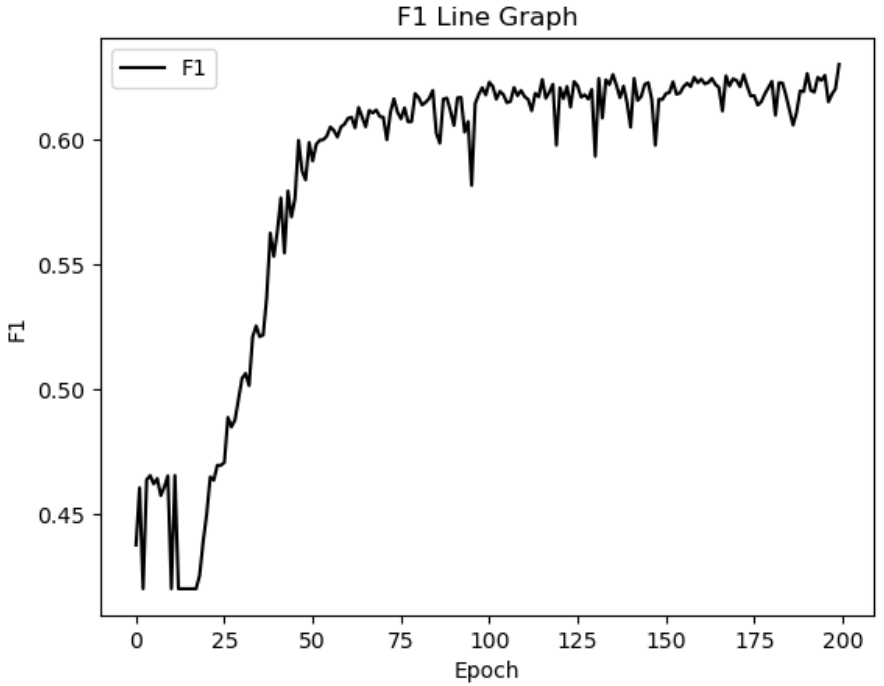
1. Recompile model before fit

Test: Run a 1 epoch model 200 times, record test accuracy

Recompile



Same Model



Good News and Bad News

Bad news:

- This *could* invalidate everything done up to this point …

Good news:

- If training the same model on different permutations of the same data frame results in better performance, then this still is a good model, just much slower to train
- All the code and theory is there, it just has to be re-run
- We can simply re-make weighted shuffle dfs to train the model instead of messing with well written, highly optimized code

The first graph is essentially a 1 epoch training each time which shows high variance

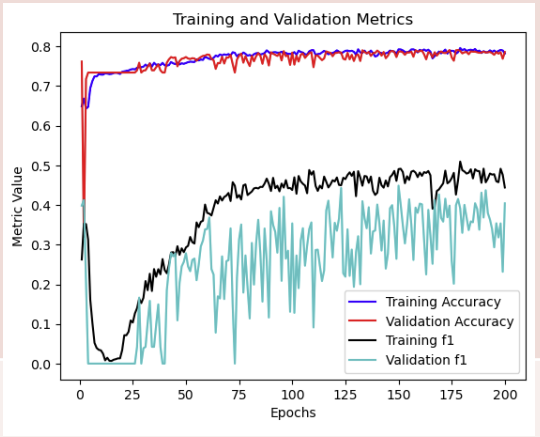
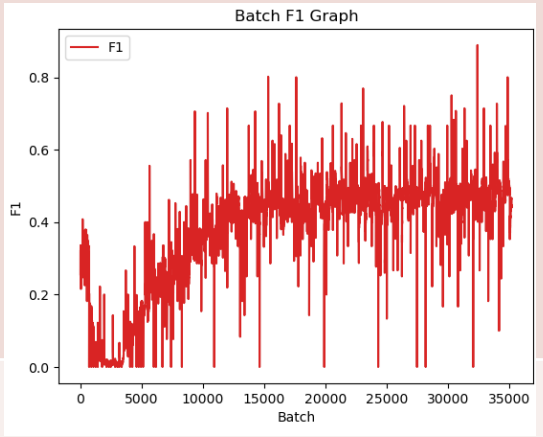
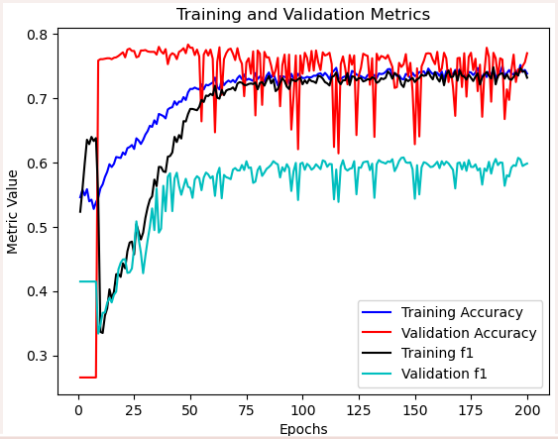
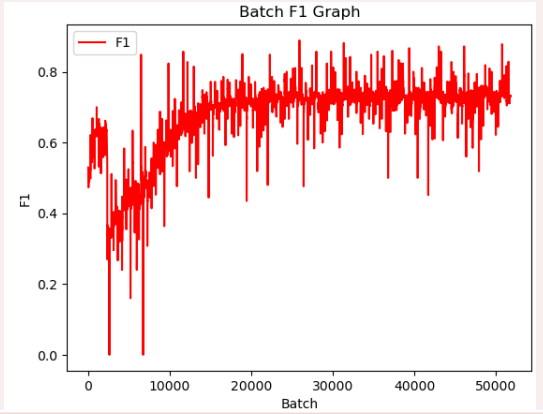
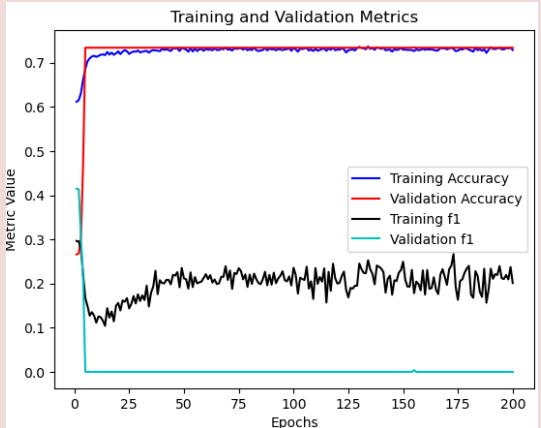
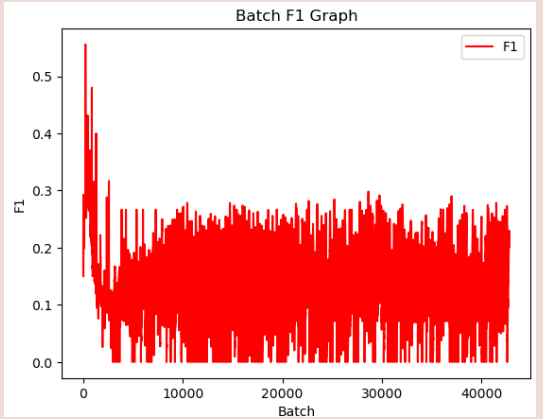
Changes to Functions

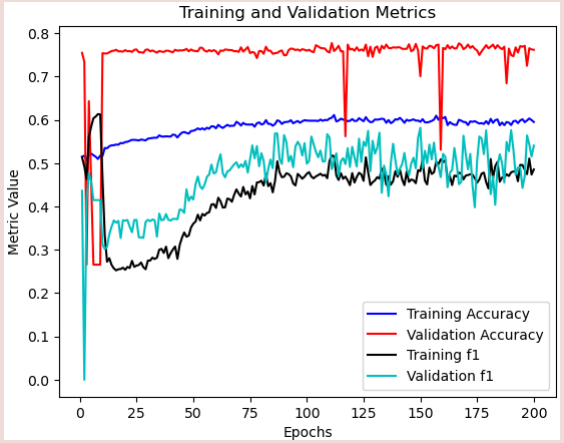
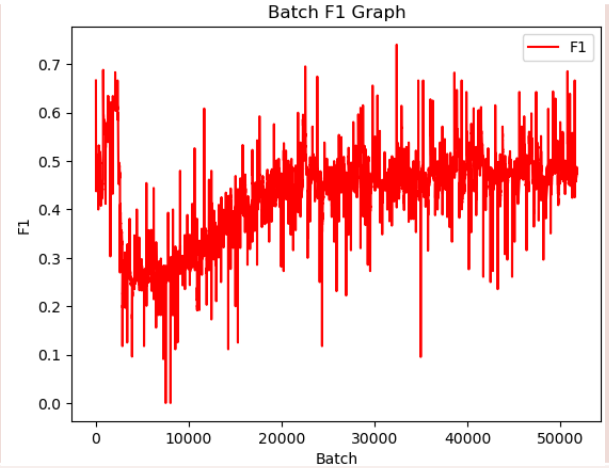
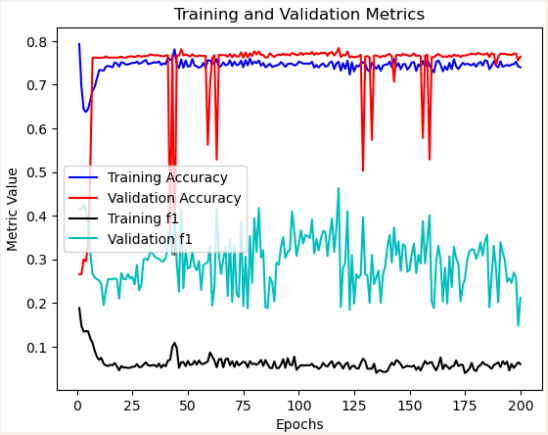
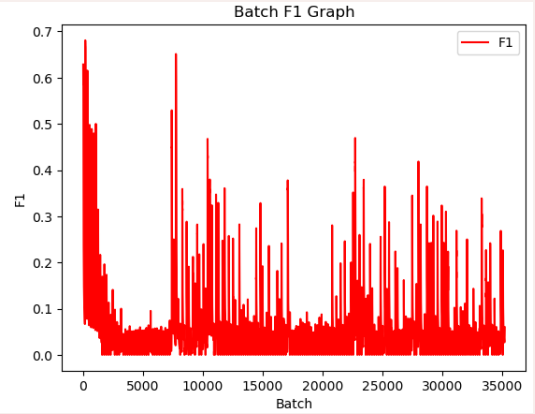
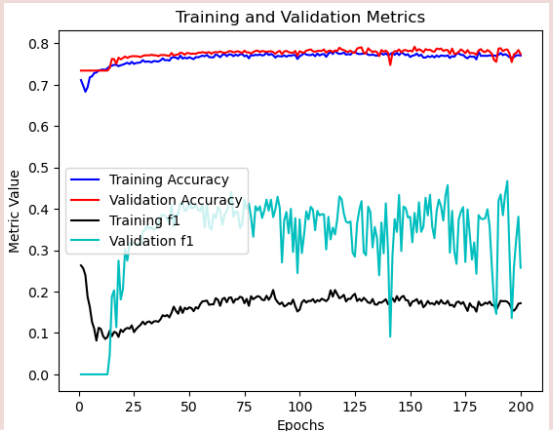
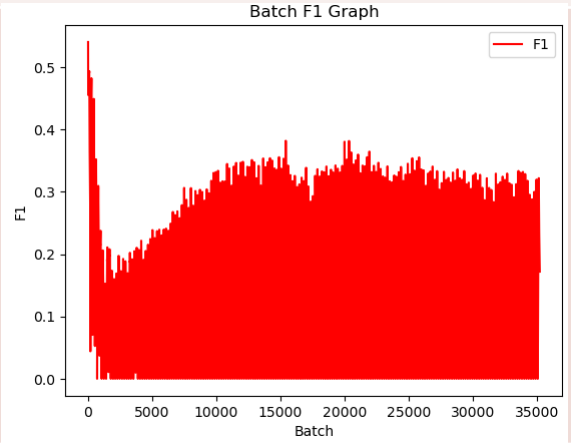
Recompile model before fitting

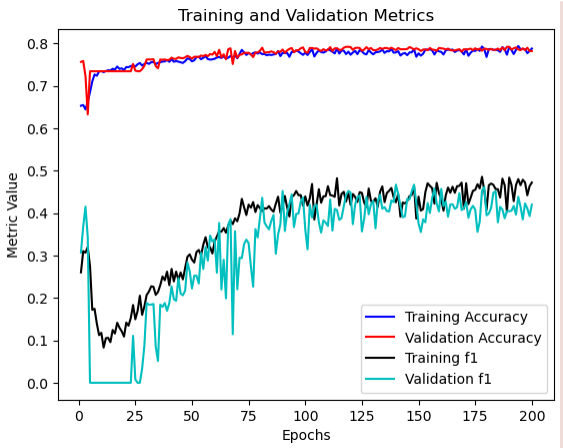
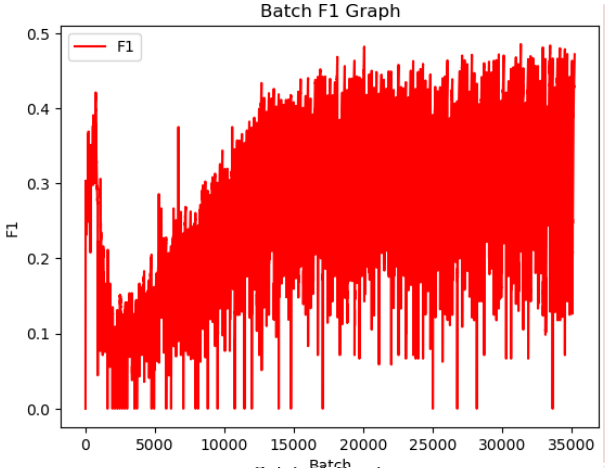
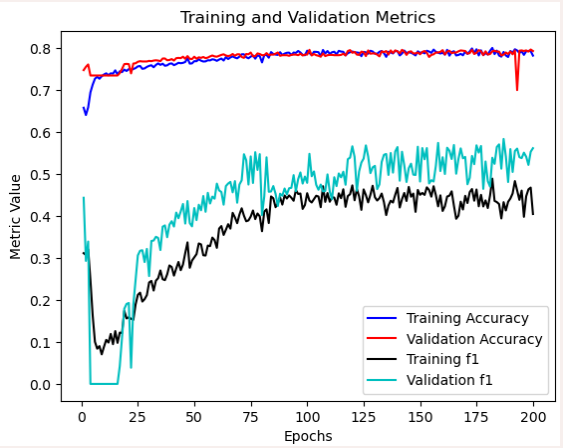
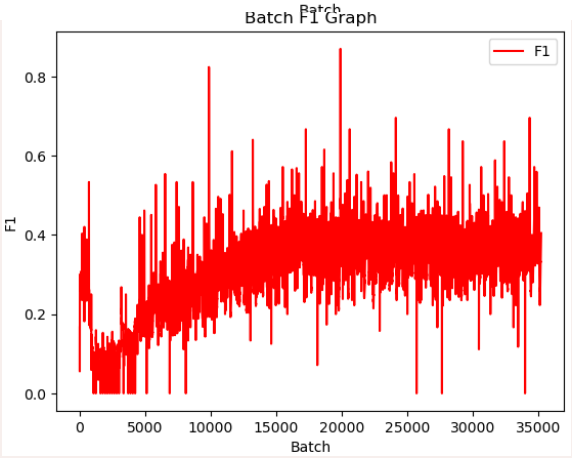
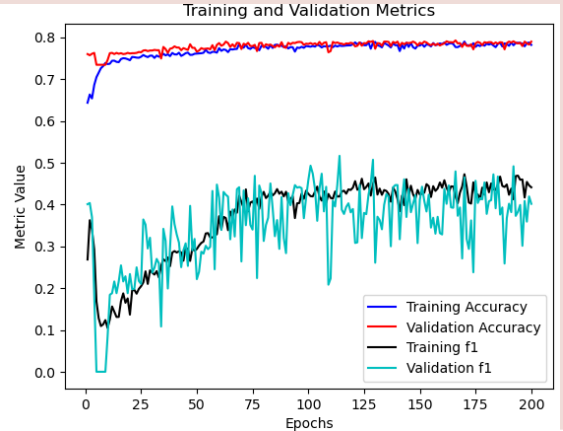
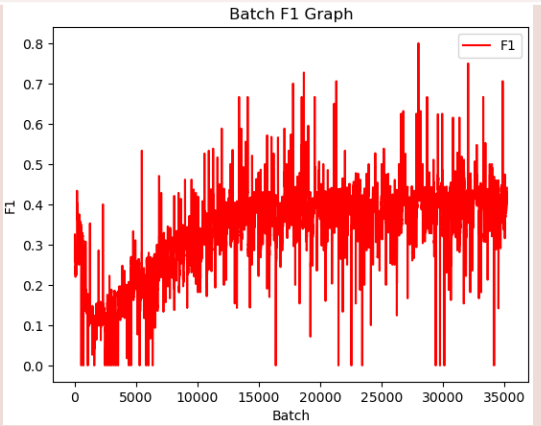
Use training threshold for everything

2. Weighted Sampled:

```
def WNN(high, low, xtest, ytest):
    # Define the model architecture
    model = Sequential()
    model.add(Dense(64, activation='relu', input_dim=30))
    model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))

    # Compile the model
    model.compile(optimizer='adam', loss='binary_crossentropy',
                  metrics=['accuracy', f1_m])

    BCP.batch_accuracy.clear()
    BCP.batch_f1.clear()

    val_data = (xtest, ytest)

    accuracy = []
    val_accuracy = []
    f1 = []
    val_f1 = []
    epochs = range(1, 201)

    for i in range(200):
        df = weightedBatches(high, low, 5, 32)
        #print(df.head(1))
        xtrain = df.drop("Churn_Yes", axis=1)
        ytrain = df["Churn_Yes"]

        history = model.fit(xtrain, ytrain, epochs=1, batch_size=32, shuffle=False,
                            validation_data=val_data, callbacks = [BCP()], verbose=0)
        accuracy.append(history.history['accuracy'])
        val_accuracy.append(history.history['val_accuracy'])
        f1.append(history.history['f1_m'])
        val_f1.append(history.history['val_f1_m'])
```

DF	Train	Test	Epoch Graph	Batch Graph
Regular SH	Confusion Matrix: [[3313 817] [447 1048]] Best Threshold: 0.31 Accuracy: 0.7752888888888889 Recall: 0.7010033444816054 F1: 0.6238095238095238	Confusion Matrix: [[825 208] [110 264]] Best Threshold: 0.31 Accuracy: 0.7739872068230277 Recall: 0.7058823529411765 F1: 0.6241134751773049		
Over SH	Confusion Matrix: [[2980 1150] [734 3396]] Best Threshold: 0.48 Accuracy: 0.7719128329297821 Recall: 0.8222760290556901 F1: 0.7828492392807747	Confusion Matrix: [[714 319] [71 303]] Best Threshold: 0.48 Accuracy: 0.7228144989339019 Recall: 0.8101604278074866 F1: 0.608433734939759		
Over-Miss	Confusion Matrix: [[968 3918] [42 1900]] Best Threshold: 0.24 Accuracy: 0.4200351493848858 Recall: 0.9783728115345005 F1: 0.4896907216494845	Confusion Matrix: [[234 799] [8 366]] Best Threshold: 0.24 Accuracy: 0.42643923240938164 Recall: 0.9786096256684492 F1: 0.4756335282651072		

DF	Train	Test	Epoch Graph	Batch Graph
Hybrid SH	Confusion Matrix: [[1457 2673] [299 3831]] Best Threshold: 0.47 Accuracy: 0.6401937046004843 Recall: 0.9276029055690073 F1: 0.7205190897122437	Confusion Matrix: [[357 676] [17 357]] Best Threshold: 0.47 Accuracy: 0.5074626865671642 Recall: 0.9545454545454546 F1: 0.5074626865671642		
HTP	Confusion Matrix: [[3004 1126] [384 1111]] Best Threshold: 0.28 Accuracy: 0.7315555555555555 Recall: 0.7431438127090301 F1: 0.5953912111468381	Confusion Matrix: [[752 281] [94 280]] Best Threshold: 0.28 Accuracy: 0.7334754797441365 Recall: 0.7486631016042781 F1: 0.5989304812834224		
SBD	Confusion Matrix: [[3212 918] [490 1005]] Best Threshold: 0.25 Accuracy: 0.7496888888888888 Recall: 0.6722408026755853 F1: 0.58806319485079	Confusion Matrix: [[800 233] [115 259]] Best Threshold: 0.25 Accuracy: 0.7526652452025586 Recall: 0.6925133689839572 F1: 0.5981524249422633		

DF	Train	Test	Epoch Graph	Batch Graph
Save	Confusion Matrix: [[3121 1009] [342 1153]] Best Threshold: 0.32 Accuracy: 0.7598222222222222 Recall: 0.7712374581939799 F1: 0.6305715066994804	Confusion Matrix: [[775 258] [94 280]] Best Threshold: 0.32 Accuracy: 0.749822316986496 Recall: 0.7486631016042781 F1: 0.6140350877192983		
Weighed Sample HTP	Confusion Matrix: [[3306 824] [429 1066]] Best Threshold: 0.38 Accuracy: 0.7772444444444444 Recall: 0.7130434782608696 F1: 0.6298375184638109	Confusion Matrix: [[813 220] [109 265]] Best Threshold: 0.38 Accuracy: 0.7661691542288557 Recall: 0.7085561497326203 F1: 0.6169965075669382		
Weighed Sample Save	Confusion Matrix: [[3197 933] [392 1103]] Best Threshold: 0.28 Accuracy: 0.7644444444444445 Recall: 0.7377926421404682 F1: 0.6247521948456528	Confusion Matrix: [[794 239] [103 271]] Best Threshold: 0.28 Accuracy: 0.7569296375266524 Recall: 0.7245989304812834 F1: 0.6131221719457014		

3. Specialized Models

Very similar performance even when model is recompiled:

```
Confusion Matrix:
[[125   6]
 [  8  64]]

True Positive: 64
True Negative: 125
False Positive: 6
False Negative: 8

Accuracy: 0.9310344827586207
True-Positve Rate: 0.8888888888888888
F1 score: 0.9014084507042254
```

```
Confusion Matrix:
[[882  20]
 [ 72 230]]

True Positive: 230
True Negative: 882
False Positive: 20
False Negative: 72

Accuracy: 0.9235880398671097
True-Positve Rate: 0.7615894039735099
F1 score: 0.8333333333333333
```

```
Total Confusion Matrix:
[[1007   26]
 [  80  294]]

Accuracy: 0.9246624022743426
True-Positve Rate: 0.786096256684492
F1 score: 0.8472622478386167
```

3. Specialized Models

Using a Gradient Boosted Tree and NN:

```
Confusion Matrix:
[[131   0]
 [  0  72]]

True Positive: 72
True Negative: 131
False Positive: 0
False Negative: 0

Accuracy: 1.0
True-Positve Rate: 1.0
F1 score: 1.0
```

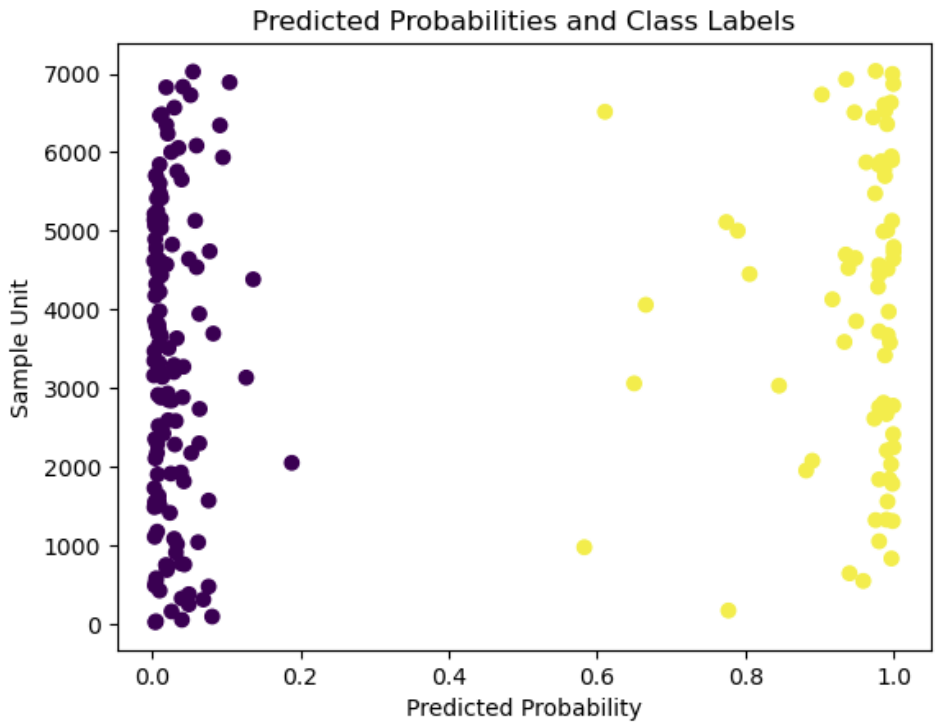
```
Confusion Matrix:
[[873  29]
 [ 63 239]]

True Positive: 239
True Negative: 873
False Positive: 29
False Negative: 63

Accuracy: 0.9235880398671097
True-Positve Rate: 0.7913907284768212
F1 score: 0.8385964912280702
```

```
Total Confusion Matrix:
[[1004   29]
 [  63  311]]

Accuracy: 0.9346126510305615
True-Positve Rate: 0.8315508021390374
F1 score: 0.8711484593837535
```

Note: Using 2 trees also had good performance, seems like specialization is the key more so than the type of model used. Also note, GBM predict() uses .5 threshold which I did not change

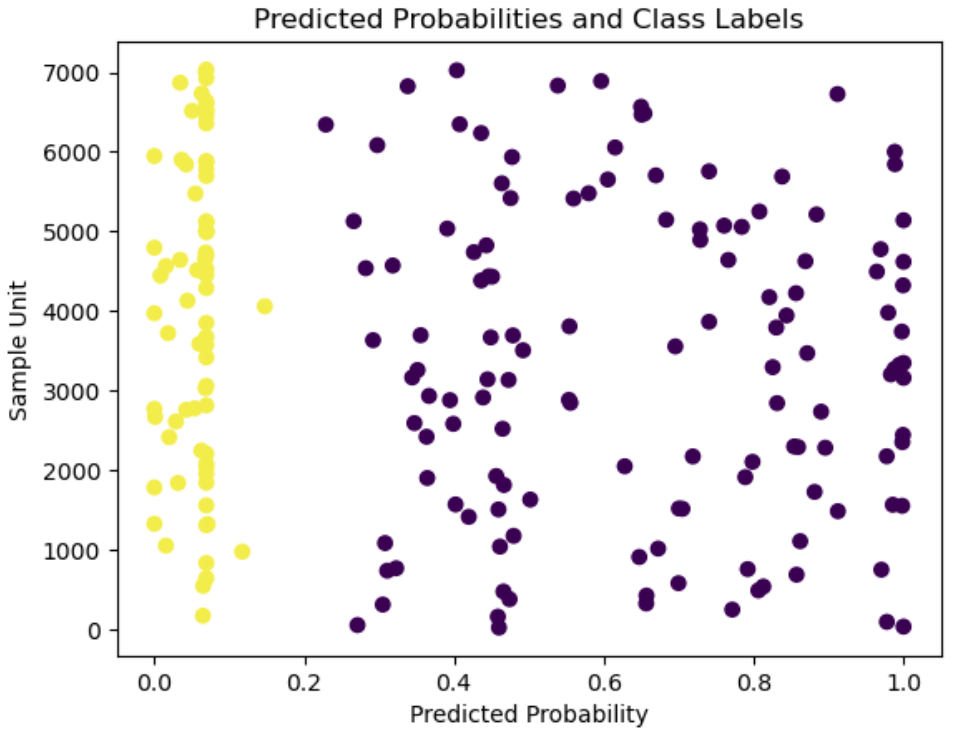
3a. Probability spread HTP

Tree



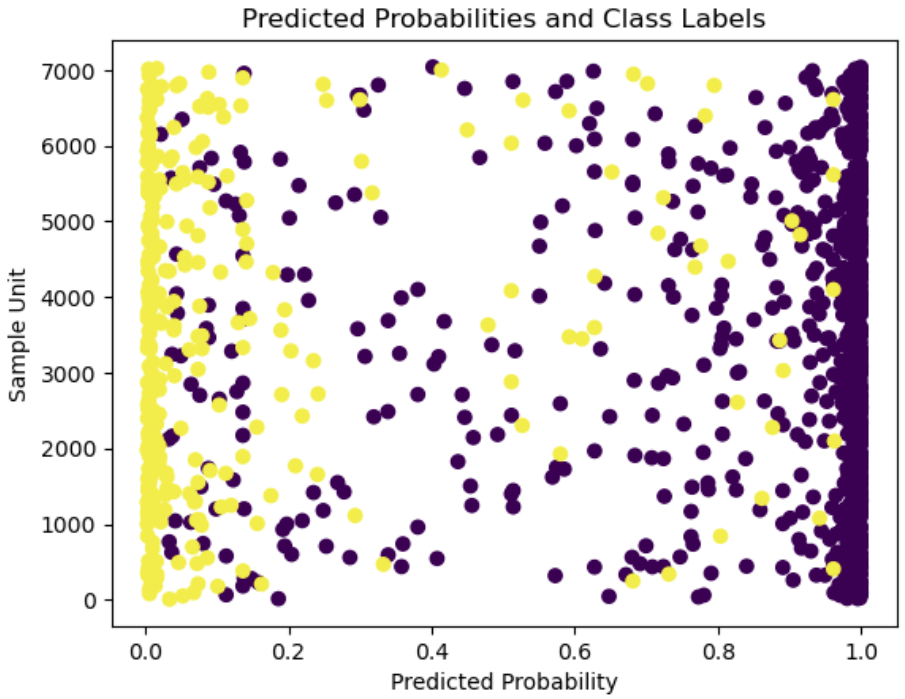
Purple = 0

NN



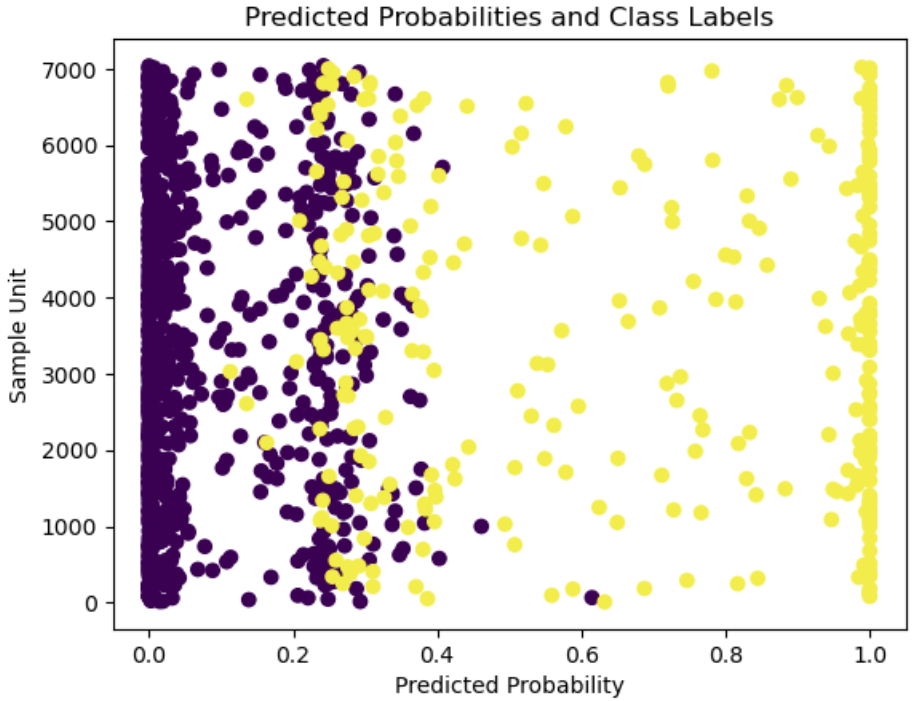
3a. Probability spread Regular

Tree



Purple = 0

NN



3a. Probability spread

The models are really bad at predicting the other dataset, so in a test set, if one model gives a particular sample unit a high probability, it really does not mean much, nor can we use the probabilities given by the two models to decide who classification to pick

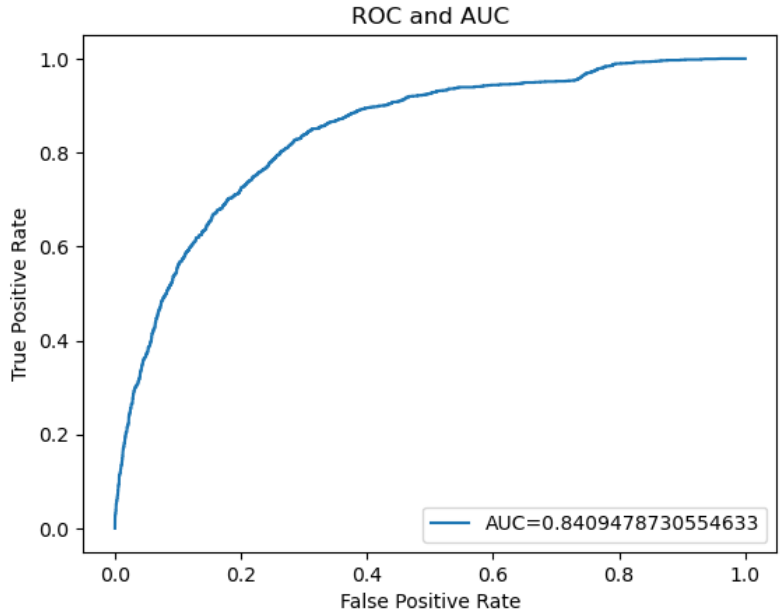
3b. Weight

This strategy applies a weight to the probabilities from the NN and tree and then combines them to find the best threshold and weight which try to combine the specialized predictions

```
def bestWeight(pred1, pred2, acc):
    w1 = 0
    w2 = 1
    f1 = 0
    t = 0
    p = 0

    for i in range(101):
        weight1 = i/100
        weight2 = 1-weight1

        pred = (pred1*weight1) + (pred2*weight2)
        test = thresh2(pred, acc)
        #print("Weight1:", weight1, "Weight2:", weight2, "Threshold:", test[1], "F

        if (test[0] > f1):
            p = pred
            f1 = test[0]
            w1 = weight1
            w2 = weight2
            t = test[1]

    print("Best Weight1:", w1)
    print("Best Weight2:", w2)
    print("Best F1:", f1)
    print("Best Threshold:", t)

    return([p, t, w1, w2])
```

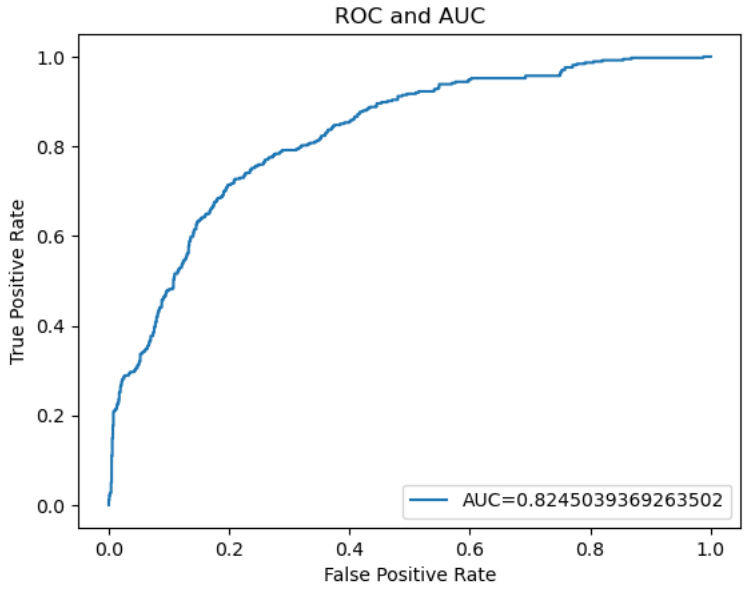
3b. Results

```
Best Weight1: 0.03
Best Weight2: 0.97
Best F1: 0.6378048780487805
Best Threshold: 0.33
```




```
Confusion Matrix:
[[3391  739]
 [ 449 1046]]
Best Threshold: 0.33
Accuracy: 0.7888
Recall: 0.6996655518394649
F1: 0.6378048780487805
```

Test




```
Confusion Matrix:
[[835 198]
 [112 262]]
Best Threshold: 0.33
Accuracy: 0.7796730632551528
Recall: 0.7005347593582888
F1: 0.6282973621103117
```

3b. Conclusion

This did not work because the HTP model is too specialized. It was so bad at predicting other sample units that it got very little weight (0.3) and so the performance was just the NN performance.

```
HTPDF = HTP.copy()
HTPDF["HTP"] = 1
notHTPDF = notHTP.copy()
notHTPDF["HTP"] = 0
HTP2 = pd.concat([HTPDF, notHTPDF])

HTPDF = HTPTest.copy()
HTPDF["HTP"] = 1
notHTPDF = notHTPTest.copy()
notHTPDF["HTP"] = 0
HTP3 = pd.concat([HTPDF, notHTPDF])
HTP3

xtrainHTP = HTP2.drop(["Churn_Yes", "Prob", "Class", "HTP"], axis = 1)
ytrainHTP = HTP2["HTP"]

xtestHTP = HTP3.drop(["Churn_Yes", "Prob", "Class", "HTP"], axis = 1)
ytestHTP = HTP3["HTP"]

gbc = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, random_state=42, max_depth = 8)
gbc.fit(xtrainHTP,ytrainHTP)
ypred = gbc.predict(xtestHTP)

HTP3["Model"] = ypred

HTPTestNN = HTP3[HTP3["Model"] == 1]

notHTPTestNN = HTP3[HTP3["Model"] == 0]

NNTree(HTP, notHTP, HTPTestNN, notHTPTestNN)
```

3c. HTP Tree

Summary: Use a NN to predict HTP units and then feed into specialized model

3c. HTP Tree Results

HTP is not very predictable by a tree, as such, the sample units separation is difficult

```
Total Confusion Matrix:
[[863 170]
 [138 236]]

Accuracy: 0.7810945273631841
True-Positve Rate: 0.6310160427807486
F1 score: 0.6051282051282051
```

4. Try another dataset

https://www.kaggle.com/datasets/mnassrib/telecom-churn-datasets?select=churn-bigml-80.csv

4. Try another dataset

HTP Tree

```
Confusion Matrix:
[[29  0]
 [ 3 29]]

True Positive: 29
True Negative: 29
False Positive: 0
False Negative: 3

Accuracy: 0.9508196721311475
True-Positve Rate: 0.90625
F1 score: 0.9508196721311475
```

Regular NN

```
Confusion Matrix:
[[510  35]
 [ 10  51]]

True Positive: 51
True Negative: 510
False Positive: 35
False Negative: 10

Accuracy: 0.9257425742574258
True-Positve Rate: 0.8360655737704918
F1 score: 0.6938775510204083
```

Before

```
Confusion Matrix:
[[572   2]
 [ 58  35]]

True Positive: 35
True Negative: 572
False Positive: 2
False Negative: 58

Accuracy: 0.9100449775112444
True-Positve Rate: 0.3763440860215054
F1 score: 0.5384615384615384
```

After

```
Total Confusion Matrix:
[[539  35]
 [ 13  80]]

Accuracy: 0.9280359820089955
True-Positve Rate: 0.8602150537634409
F1 score: 0.7692307692307692
```

Conclusion

Specialized models seem to work good in theory, but trying to separate the test set is difficult. Could there be some way of creating specialized models that works with a real test set?(Probably (pie charts), but it will likely be very situational)

Week 7

To-Do

1. centroid using numbers and categories
2. 3 neural networks (or tree) (SBD)
3. tie breaker neural network trained on everything
4. Listen to who is more certain

4. Listen to who is more certain

HTP

```
Confusion Matrix:
[[233 800]
 [265 109]]
Best Threshold: 0.5
Accuracy: 0.24307036247334754
Recall: 0.2914438502673797
F1: 0.16991426344505065
```

Reg

```
Confusion Matrix:
[[841 192]
 [119 255]]
Best Threshold: 0.29
Accuracy: 0.7789623312011372
Recall: 0.6818181818181818
F1: 0.6211936662606576
```

Total

```
Confusion Matrix:
[[1018   15]
 [ 363   11]]
Best Threshold: 0.29
Accuracy: 0.7313432835820896
Recall: 0.029411764705882353
F1: 0.05499999999999999
```

3. tie breaker neural network trained on everything

HTP

```
Confusion Matrix:
[[233 800]
 [265 109]]
Best Threshold: 0.5
Accuracy: 0.2430703624733475 4
Recall: 0.2914438502673797
F1: 0.16991426344505065
```

Reg

```
Confusion Matrix:
[[833 200]
 [107 267]]
Best Threshold: 0.27
Accuracy: 0.7818052594171997
Recall: 0.713903743315508
F1: 0.6349583828775268
```

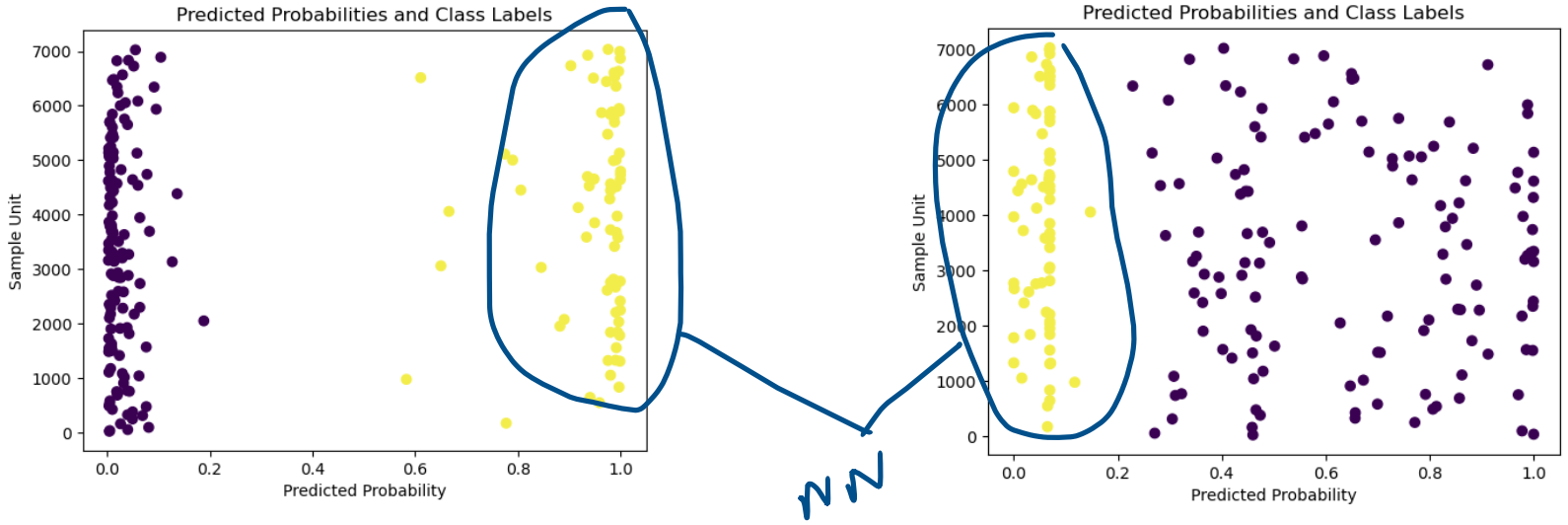
Every

```
Confusion Matrix:
[[825 208]
 [110 264]]
Best Threshold: 0.32
Accuracy: 0.7739872068230277
Recall: 0.7058823529411765
F1: 0.6241134751773049
```

Total

```
Confusion Matrix:
[[834 199]
 [113 261]]
Best Threshold: n/a
Accuracy: 0.778251599147121 5
Recall: 0.6978609625668449
F1: 0.6258992805755396
```

2. 3 neural networks (or tree) (SBD)



NN

```
[[746 118]
 [ 91 188]]
```

```
Confusion Matrix:
[[440  37]
 [ 46  12]]
Best Threshold: 0.12
Accuracy: 0.8448598130841122
Recall: 0.20689655172413793
F1: 0.22429906542056074
```

Other

```
def Triple(df, df2, total, test):

    xColumns = list(probdf.columns)[:-3]

    xtrain1 = df[xColumns]
    ytrain1 = df["Churn_Yes"]

    xtrain2 = df2[xColumns]
    ytrain2 = df2["Churn_Yes"]

    xtotal = total[xColumns]
    ytotal = total["Churn_Yes"]

    xtest = test[xColumns]
    ytest = test["Churn_Yes"]

    tp = mt(xtrain1, ytrain1, xtotal, xtest)
    nnp = mm(xtrain1, ytrain1, xtotal, xtest, ytotal)

    dptrain = mdp(total, tp[0], nnp[0])

    dptest = mdp(test, tp[1], nnp[1])

    dpn = mm(dptrain[0], dptrain[1], dptest[0], dptest[0], dptest[1])

    bestT = thresh(dpn[1], dptest[1])

    classPred = [0 if val < bestT else 1 for val in dpn[1]]

    evaluate(dptest[1], classPred, bestT)

    pred = tp[2].predict_proba(dptest[2])[:,1]
    ypred = nnp[2].predict(dptest[2])

    combine =  [max(x, y, key=lambda num: abs(num - 50)) for x, y in zip(
    bestT = thresh(combine, dptest[3])

    evalNN(bestT, combine, dptest[3])
```

1. Kmed

Idea1 (idea on to-do list): use kmed. to cluster HTP units into one cluster. Also, cluster notHTP units into one cluster. Then calculate the distance of the test units to the middle of each cluster. Which ever cluster the test unit is closer to will be the model that predicts the sample unit.

Idea2: Make clusters of entire dataset (2-4). Make a corresponding model for each sample cluster. Cluster fit test units and feed to closest model for prediction. This method removes the reliance on the arbitrary “HTP” distinction. (Next week?)

```
def cluster (df, df2, test):
    xColumns = list(probdf.columns)[:-3]

    xtrain1 = df[xColumns]
    ytrain1 = df["Churn_Yes"]

    xtrain2 = df2[xColumns]
    ytrain2 = df2["Churn_Yes"]

    xtest = test[xColumns]
    ytest = test["Churn_Yes"]

    k1 = KMedoids(n_clusters=1)
    k1.fit(xtrain1)

    k2 = KMedoids(n_clusters=1)
    k2.fit(xtrain2)

    d1 = k1.transform(xtest)

    d2 = k2.transform(xtest)

    clusterdf = test.copy()
    clusterdf["d1"] = d1
    clusterdf["d2"] = d2

    k1test = clusterdf[clusterdf["d1"] < clusterdf["d2"]]
    print("Predicted HTPTest:", len(k1test))
    print("Acutal HTPTest:", len(HTPTest))

    k2test = clusterdf[clusterdf["d1"] >= clusterdf["d2"]]
    print("Predicted notHTPTest:", len(k2test))
    print("Acutal notHTPTest:", len(notHTPTest))
```

```
cluster(HTP, notHTP, test)
```

```
Predicted HTPTest: 629
Acutal HTPTest: 203
Predicted notHTPTest: 778
Acutal notHTPTest: 1204
```

1. Kmed

1. Kmed

HTP

```
Confusion Matrix:
[[126 274]
 [188  41]]

True Positive: 41
True Negative: 126
False Positive: 274
False Negative: 188
```

Reg

```
Confusion Matrix:
[[512 121]
 [ 57  88]]

True Positive: 88
True Negative: 512
False Positive: 121
False Negative: 57
```

Total

```
Total Confusion Matrix:
[[638 395]
 [245 129]]

Accuracy: 0.5451314854299929
True-Positve Rate: 0.3449197860962567
F1 score: 0.2873051224944321
```

1. Kmed

Why did it not work? HTP units were closer to not HTP units

CM showing predicted HTP vs real HTP

```
Confusion Matrix:
[[674 530]
 [104  99]]

True Positive: 99
True Negative: 674
False Positive: 530
False Negative: 104

Accuracy: 0.5493958777540867
True-Positve Rate: 0.4876847290640394
F1 score: 0.23798076923076925

array([[674, 530],
       [104,  99]], dtype=int64)
```

More possibilities

We could work with the distance from the medoids to assign which model handles the sample unit. For example, if a sample unit is within x distance from the HTP cluster medoid, the it is predicted by the HTP model (repeat for n clusters). Any sample unit that does not fit in any category is predicted by a NN trained on everything.

We could try method 2: random clusters (removes reliance on “HTP” label)

Week 8

To-Do

1. Threshold HTP network
2. Random cluster
3. Threshold distance
4. Retry different probability NN*
5. Use more complex NN as benchmark
6. Try using simpler models, see if you get consistent results

2. Random cluster

Cluster: 1

```
22/22 [==============================]
Cluster 1 Training
Confusion Matrix:
[[  0 615]
 [  0  87]]

Accuracy: 0.12393162393162394
True-Positve Rate: 1.0
F1 score: 0.22053231939163498
Best Threshold: 0.01

6/6 [==============================]
Cluster 1 Testing
Confusion Matrix:
[[  0 149]
 [  0  25]]

Accuracy: 0.14367816091954022
True-Positve Rate: 1.0
F1 score: 0.25125628140703515
Best Threshold: 0.01
```

Cluster: 2

```
27/27 [==============================]
Cluster 2 Training
Confusion Matrix:
[[310 313]
 [ 41 172]]

Accuracy: 0.5765550239234449
True-Positve Rate: 0.8075117370
F1 score: 0.4928366762177651
Best Threshold: 0.27

7/7 [==============================]
Cluster 2 Testing
Confusion Matrix:
[[70 79]
 [14 37]]

Accuracy: 0.535
True-Positve Rate: 0.7254901960
F1 score: 0.44311377245508987
Best Threshold: 0.27
```

Cluster: 3

```
43/43 [==============================]
Cluster 3 Training
Confusion Matrix:
[[897 160]
 [ 96 208]]

Accuracy: 0.8119030124908155
True-Positve Rate: 0.684210526
F1 score: 0.6190476190476191
Best Threshold: 0.36

11/11 [==============================]
Cluster 3 Testing
Confusion Matrix:
[[218  45]
 [ 32  53]]

Accuracy: 0.7787356321839081
True-Positve Rate: 0.623529411
F1 score: 0.5792349726775956
Best Threshold: 0.36
```

Cluster: 4

```
25/25 [==============================]
Cluster 4 Training
Confusion Matrix:
[[  0 640]
 [  0 133]]

Accuracy: 0.17205692108667528
True-Positve Rate: 1.0
F1 score: 0.29359823399558493
Best Threshold: 0.01

7/7 [==============================]
Cluster 4 Testing
Confusion Matrix:
[[  0 181]
 [  0  27]]

Accuracy: 0.12980769230769232
True-Positve Rate: 1.0
F1 score: 0.22978723404255322
Best Threshold: 0.01
```

Cluster: 5

```
62/62 [==============================]
Cluster 5 Training
Confusion Matrix:
[[918 277]
 [178 580]]

Accuracy: 0.7670250896057348
True-Positve Rate: 0.7651715039577837
F1 score: 0.7182662538699689
Best Threshold: 0.44

15/15 [==============================]
Cluster 5 Testing
Confusion Matrix:
[[222  69]
 [ 51 135]]

Accuracy: 0.7484276729559748
True-Positve Rate: 0.7258064516129032
F1 score: 0.6923076923076922
Best Threshold: 0.44
```

Total Confusion Matrix:

```
[[510 523]
 [ 97 277]]

Accuracy: 0.5593461265103056
True-Positve Rate: 0.7406417112299465
F1 score: 0.4718909710391822
```

Didn’t work.

2. Random cluster

Some cluster probabilities were too similar

(The ones that had poor performance)

```
 Cluster: 4
14/14 [==========
[[0.22532095]
 [0.22784157]
 [0.22894892]
 [0.22885603]
 [0.22485216]
 [0.2298614 ]
 [0.22567743]
 [0.22855172]
 [0.22531821]
 [0.22685385]
 [0.22467807]
 [0.22533876]
 [0.23127928]]
```

```
 Cluster: 1
15/15 [==========
[[0.32098222]
 [0.32098222]
 [0.3210272 ]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]
 [0.32098222]]
```

5. Use more complex NN as benchmark

More complex

```
Confusion Matrix:
[[798 235]
 [ 99 275]]
Best Threshold: 0.36
Accuracy: 0.7626154939587776
Recall: 0.7352941176470589
F1: 0.6221719457013575
```

```
# Define the model architecture
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=30))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
```

Current

```
Confusion Matrix:
[[831 202]
 [114 260]]
Best Threshold: 0.31
Accuracy: 0.775408670931059
Recall: 0.6951871657754011
F1: 0.6220095693779905
```

```
# Define the model architecture
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=30))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
```

Less Complex (32 neurons)

```
Confusion Matrix:
[[771 262]
 [ 94 280]]
Best Threshold: 0.32
Accuracy: 0.7469793887704336
Recall: 0.7486631016042781
F1: 0.611353711790393
```

128 Neurons

```
Confusion Matrix:
[[811 222]
 [104 270]]
Best Threshold: 0.28
Accuracy: 0.7683013503909026
Recall: 0.7219251336898396
F1: 0.6235565819861433
```

4. Retry different probability NNs

HTP

```
23/23 [==============================]
Confusion Matrix:
[[433  12]
 [ 12 249]]

Accuracy: 0.9660056657223796
True-Positve Rate: 0.9540229885057471
F1 score: 0.9540229885057472
Best Threshold: 0.38

44/44 [==============================]
Confusion Matrix:
[[234 799]
 [264 110]]

Accuracy: 0.24449182658137883
True-Positve Rate: 0.29411764705882354
F1 score: 0.17147310989867498
Best Threshold: 0.38
```

Not HTP

```
Confusion Matrix:
[[3482  203]
 [ 249  985]]

Accuracy: 0.9081114047570644
True-Positve Rate: 0.7982171799027553
F1 score: 0.8133773740710157
Best Threshold: 0.3

44/44 [==============================]
Confusion Matrix:
[[869 164]
 [133 241]]

Accuracy: 0.7889125799573561
True-Positve Rate: 0.6443850267379679
F1 score: 0.6187419768934532
Best Threshold: 0.3
```

DP

```
DP units in training set: 4313
DP units in test set: 1047
135/135 [==============================]
Confusion Matrix:
[[2901  353]
 [ 304  755]]

Accuracy: 0.8476698353814051
True-Positve Rate: 0.7129367327667611
F1 score: 0.6968158744808491
Best Threshold: 0.38

44/44 [==============================]
Confusion Matrix:
[[792 241]
 [104 270]]

Accuracy: 0.7547974413646056
True-Positve Rate: 0.7219251336898396
F1 score: 0.6101694915254238
Best Threshold: 0.38
```

Total

```
Confusion Matrix:
[[505 299]
 [160  83]]

Accuracy: 0.5616045845272206
True-Positve Rate: 0.34156378600823045
F1 score: 0.2656
Confusion Matrix:
[[219  10]
 [128   3]]

Accuracy: 0.6166666666666667
True-Positve Rate: 0.022900763358778626
F1 score: 0.04166666666666667

Total Confusion Matrix:
[[724 309]
 [288  86]]

Accuracy: 0.5756929637526652
True-Positve Rate: 0.22994652406417113
F1 score: 0.22366710013003901
```

6. Try using simpler models, see if you get consistent results

Tree HTP

```
Confusion Matrix:
[[131   0]
 [  0  72]]

Accuracy: 1.0
True-Positve Rate: 1.0
F1 score: 1.0
```

Log HTP

```
Confusion Matrix:
[[131   0]
 [  0  72]]

Accuracy: 1.0
True-Positve Rate: 1.0
F1 score: 1.0
```

NN not HTP

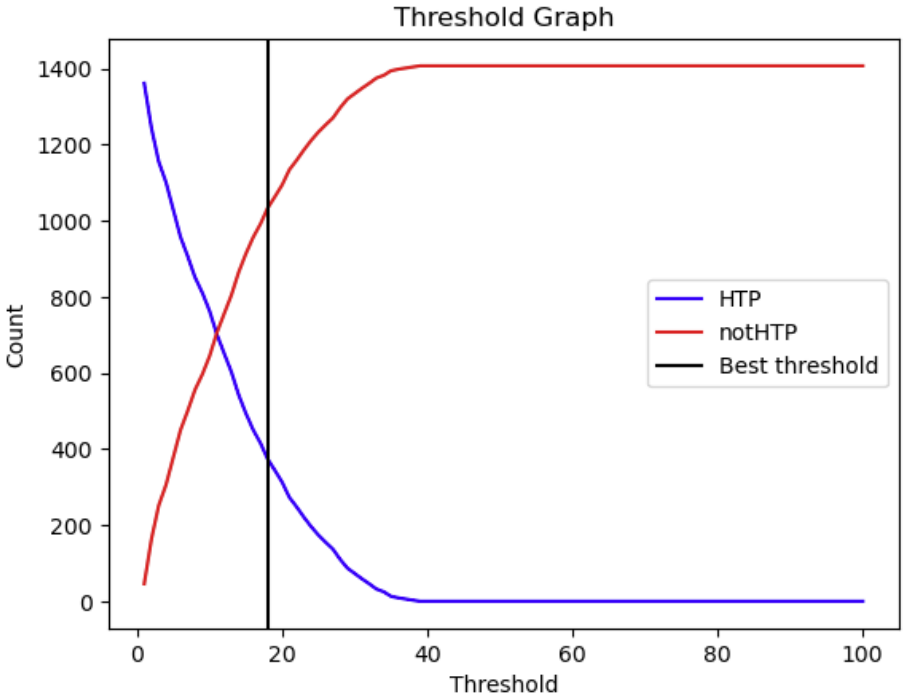
```
Confusion Matrix:
[[851  51]
 [ 49 253]]

True Positive: 253
True Negative: 851
False Positive: 51
False Negative: 49

Accuracy: 0.9169435215946844
True-Positve Rate: 0.8377483443708609
F1 score: 0.834983498349835
```

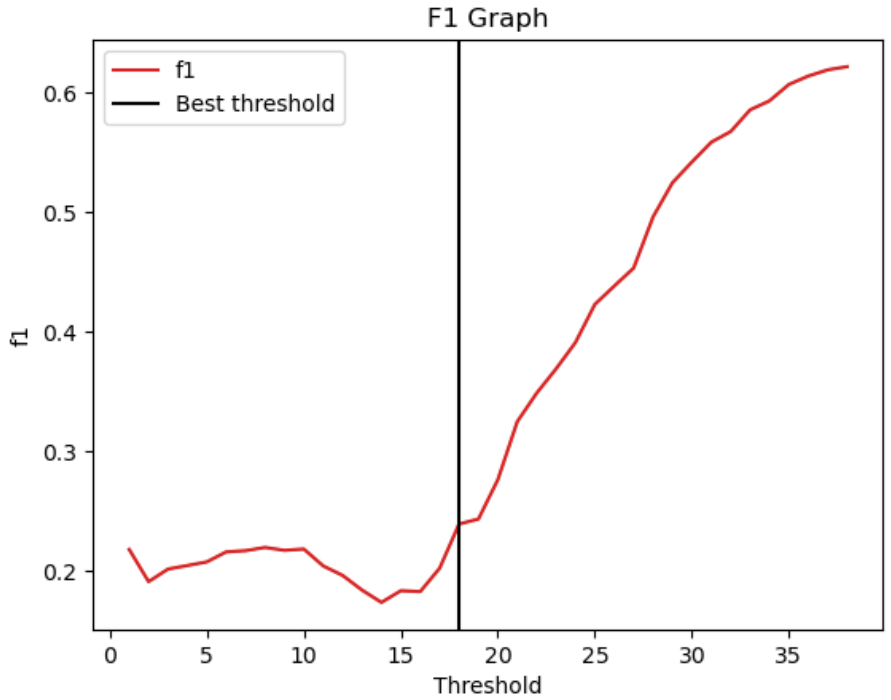
Log not HTP

```
Confusion Matrix:
[[894   8]
 [ 90 212]]

Accuracy: 0.9186046511627907
True-Positve Rate: 0.7019867549668874
F1 score: 0.8122605363984674
```

1. Threshold HTP network

Black line shows best threshold for predicting which units are HTP

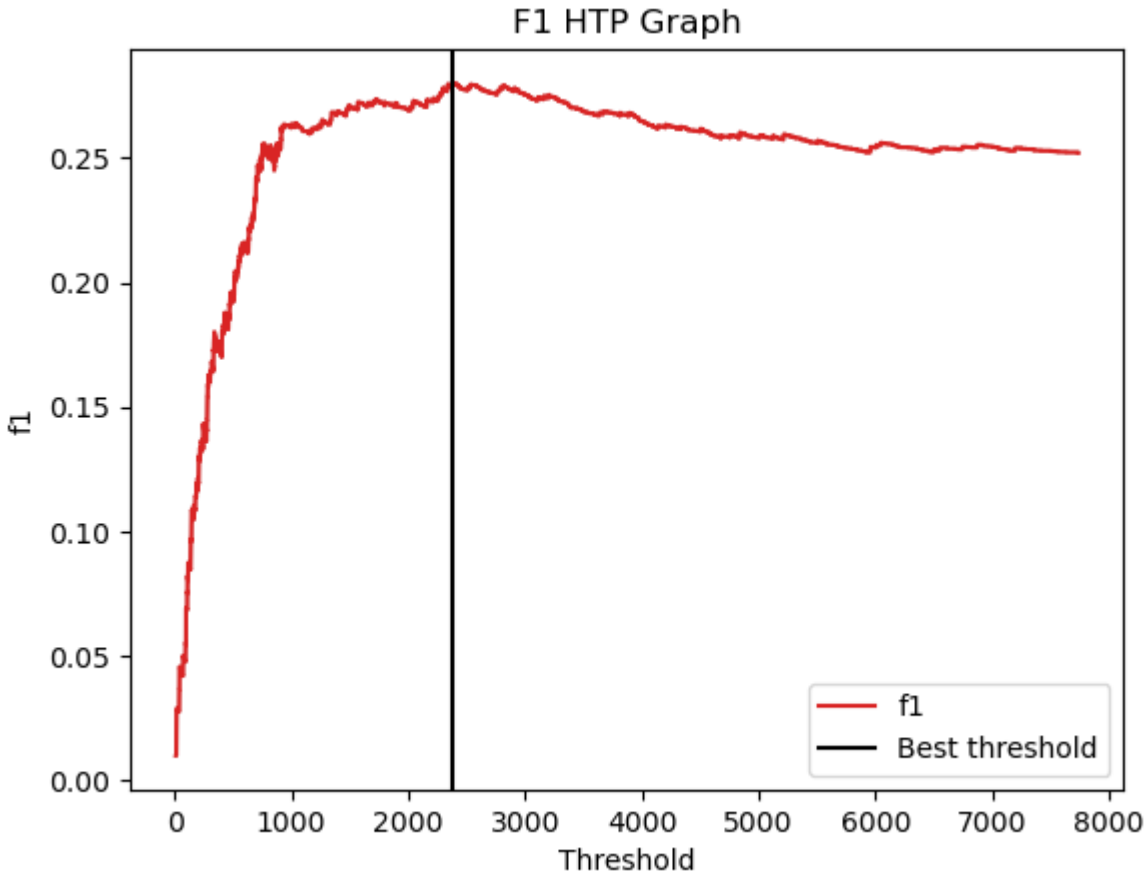




As threshold increases, the number of notHTP increases and f1 increases, meaning the best f1 is when no HTP model is being used

3. Threshold distance

The ability of clustering to predict HTP is about the same as the NN (not good)



Week 9

To-Do

1. Train on just notHTP and see if it can perform better than original (different test sets)
2. Oversample HTP-predicting network
3. Use most important variables from tree as proxy to determine which are HTP
4. standardize cont.
5. try k nearest (if can’t use cat, use just num vars)

1. Train on just notHTP and see if it can perform better than original (different test sets)

```
notHTPModel = MakeNN(notHTP[xColumns], notHTP["Churn_Yes"], test[xColumns], test["Churn_Yes"])

154/154 [==============================] - 0s 824us/step
44/44 [==============================] - 0s 949us/step
Confusion Matrix:
[[880 153]
 [152 222]]

Accuracy: 0.783226723525231
True-Positve Rate: 0.5935828877005348
F1 score: 0.5927903871829105
```

```
HTPModel = MakeNN(probdf[xColumns], probdf["Churn_Yes"], test[xColumns], test["Churn_Yes"])

176/176 [==============================] - 0s 756us/step
44/44 [==============================] - 0s 503us/step
Confusion Matrix:
[[834 199]
 [118 256]]

Accuracy: 0.7746979388770433
True-Positve Rate: 0.6844919786096256
F1 score: 0.617611580217129
```

```
notHTPModel = MakeNN(notHTP[xColumns], notHTP["Churn_Yes"], test[xColumns], test["Churn_Yes"])

153/153 [==============================] - 0s 829us/step
44/44 [==============================] - 0s 942us/step
Confusion Matrix:
[[884 137]
 [159 227]]

Accuracy: 0.7896233120113717
True-Positve Rate: 0.5880829015544041
F1 score: 0.6053333333333334
```

```
HTPModel = MakeNN(probdf[xColumns], probdf["Churn_Yes"], test[xColumns], test["Churn_Yes"])

176/176 [==============================] - 0s 871us/step
44/44 [==============================] - 0s 867us/step
Confusion Matrix:
[[765 256]
 [112 274]]

Accuracy: 0.738450604122246
True-Positve Rate: 0.7098445595854922
F1 score: 0.5982532751091703
```

```
notHTPModel = MakeNN(notHTP[xColumns], notHTP["Churn_Yes"], test[xColumns], test["Churn_Yes"])

153/153 [==============================] - 0s 859us/step
44/44 [==============================] - 0s 1ms/step
Confusion Matrix:
[[867 153]
 [132 255]]

Accuracy: 0.7974413646055437
True-Positve Rate: 0.6589147286821705
F1 score: 0.6415094339622641
```

```
HTPModel = MakeNN(probdf[xColumns], probdf["Churn_Yes"], test[xColumns], test["Churn_Yes"])

176/176 [==============================] - 0s 745us/step
44/44 [==============================] - 0s 780us/step
Confusion Matrix:
[[776 244]
 [ 90 297]]

Accuracy: 0.7626154939587776
True-Positve Rate: 0.7674418604651163
F1 score: 0.6400862068965517
```

```
notHTPModel = MakeNN(notHTP[xColumns], notHTP["Churn_Yes"], test[xColumns], test["Churn_Yes"])

151/151 [==============================] - 0s 842us/step
44/44 [==============================] - 0s 725us/step
Confusion Matrix:
[[873 177]
 [115 242]]

Accuracy: 0.7924662402274343
True-Positve Rate: 0.6778711484593838
F1 score: 0.6237113402061856
```

```
HTPModel = MakeNN(probdf[xColumns], probdf["Churn_Yes"], test[xColumns], test["Churn_Yes"])

176/176 [==============================] - 0s 797us/step
44/44 [==============================] - 0s 870us/step
Confusion Matrix:
[[807 243]
 [ 83 274]]

Accuracy: 0.7683013503909026
True-Positve Rate: 0.7675070028011205
F1 score: 0.6270022883295194
```

```
notHTPModel = MakeNN(notHTP[xColumns], notHTP["Churn_Yes"], test[xColumns], test["Churn_Yes"])

154/154 [==============================] - 0s 853us/step
44/44 [==============================] - 0s 941us/step
Confusion Matrix:
[[844 169]
 [147 247]]

Accuracy: 0.775408670931059
True-Positve Rate: 0.6269035532994924
F1 score: 0.6098765432098766
```

```
HTPModel = MakeNN(probdf[xColumns], probdf["Churn_Yes"], test[xColumns], test["Churn_Yes"])

176/176 [==============================] - 0s 807us/step
44/44 [==============================] - 0s 835us/step
Confusion Matrix:
[[817 196]
 [136 258]]

Accuracy: 0.7640369580668088
True-Positve Rate: 0.6548223350253807
F1 score: 0.6084905660377358
```

```
notHTPModel = MakeNN(notHTP[xColumns], notHTP["Churn_Yes"], test[xColumns], test["Churn_Yes"])

152/152 [==============================] - 0s 711us/step
44/44 [==============================] - 0s 872us/step
Confusion Matrix:
[[900 135]
 [143 229]]

Accuracy: 0.8024164889836531
True-Positve Rate: 0.6155913978494624
F1 score: 0.6222826086956521
```

```
HTPModel = MakeNN(probdf[xColumns], probdf["Churn_Yes"], test[xColumns], test["Churn_Yes"])

176/176 [==============================] - 0s 804us/step
44/44 [==============================] - 0s 852us/step
Confusion Matrix:
[[900 135]
 [160 212]]

Accuracy: 0.7903340440653873
True-Positve Rate: 0.5698924731182796
F1 score: 0.5897079276773296
```

4. standardize cont.

```python
from sklearn.preprocessing import StandardScaler

# Select the numerical columns to be standardized
numerical_cols = ['tenure', 'MonthlyCharges', 'TotalCharges']

# Initialize the StandardScaler
scaler = StandardScaler()

# Fit and transform the selected columns
clean[numerical_cols] = scaler.fit_transform(clean[numerical_cols])
```

```python
notHTPModel = MakeNN(notHTP[xColumns], notHTP["Churn_Yes"], test[xColumns], test["Churn_Yes"])
```

```
152/152 [==============================] - 0s 782us/step
44/44 [==============================] - 0s 500us/step
Confusion Matrix:
[[807 226]
 [111 263]]

Accuracy: 0.7604832977967306
True-Positve Rate: 0.7032085561497327
F1 score: 0.6095017381228274
```

```python
HTPModel = MakeNN(probdf[xColumns], probdf["Churn_Yes"], test[xColumns], test["Churn_Yes"])
```

```
176/176 [==============================] - 0s 745us/step
44/44 [==============================] - 0s 559us/step
Confusion Matrix:
[[864 169]
 [154 220]]

Accuracy: 0.7704335465529495
True-Positve Rate: 0.5882352941176471
F1 score: 0.5766710353866319
```

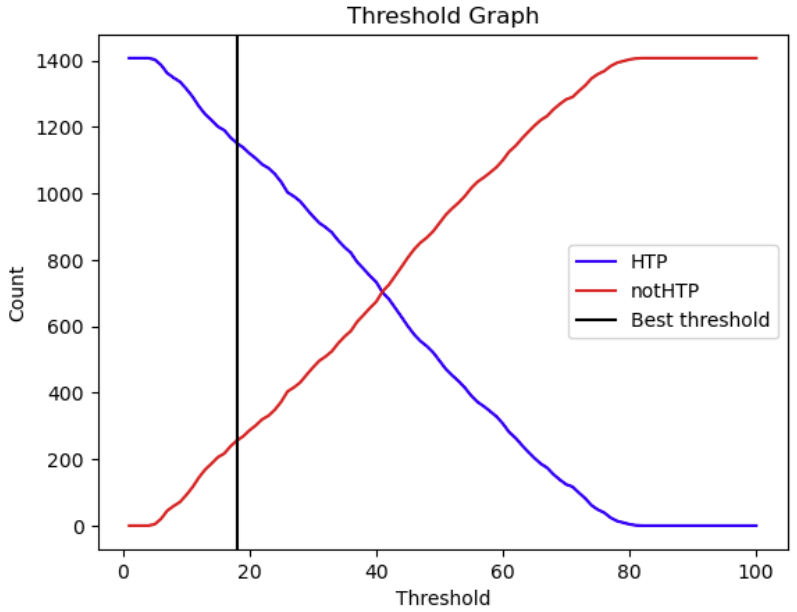
Got worse?

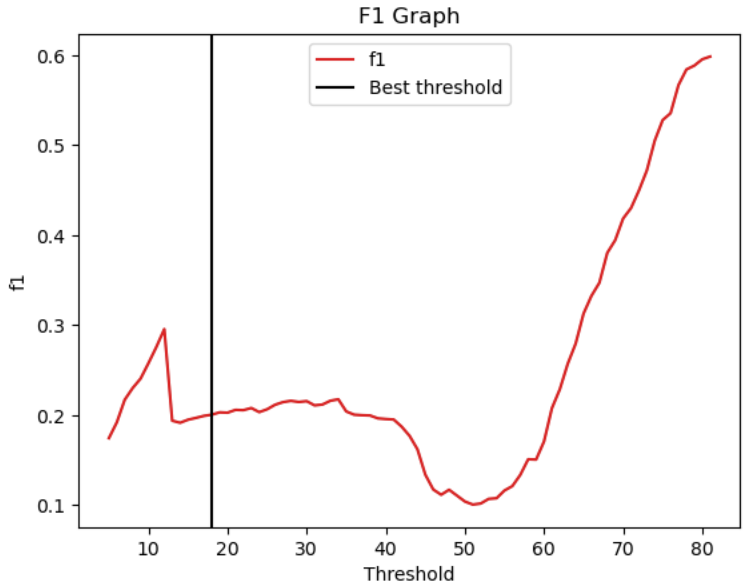
2. Oversample HTP-predicting network

```
HTP["HTP"] = 1
notHTP["HTP"] = 0
DF_HTP = pd.concat([HTP, HTP, HTP, notHTP, HTP, HTP, HTP])

print(len(DF_HTP[DF_HTP["HTP"] == 1]))
print(len(DF_HTP[DF_HTP["HTP"] == 0]))
```

```
4236
4919
```

```
BestThresh: 0.33
Confusion Matrix:
[[498 706]
 [ 27 176]]

Accuracy: 0.47903340440653874
True-Positve Rate: 0.8669950738916257
F1 score: 0.32442396313364064
```





Did not Work

5. try k nearest (if can't use cat, use just num vars)

HTP predictive power is still Bad with KNN

```
from sklearn.neighbors import KNeighborsClassifier\

HTPTest["HTP"] = 1
notHTPTest["HTP"] = 0

DF_HTP_Test = pd.concat([HTPTest, notHTPTest])

kmedoids = KNeighborsClassifier()

kmedoids.fit(DF_HTP[xColumns], DF_HTP["HTP"])

preds = kmedoids.predict(test[xColumns])

evaluate(DF_HTP_Test["HTP"], preds)
```

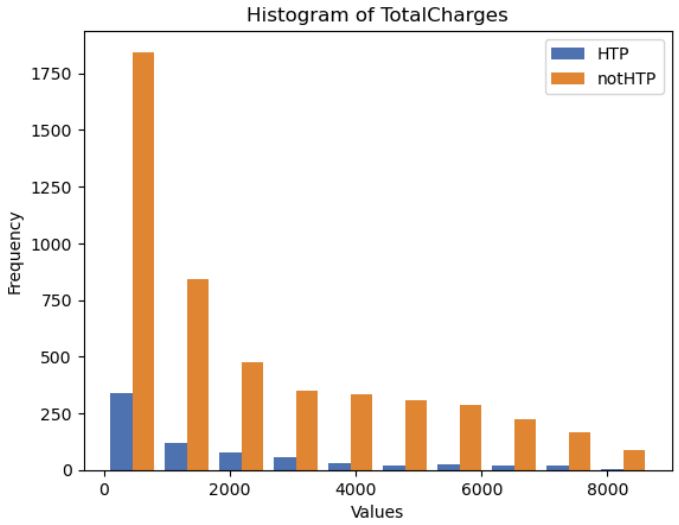
```
Confusion Matrix:
[[1165   39]
 [ 194    9]]

Accuracy: 0.8343994314143568
True-Positve Rate: 0.04433497536945813
F1 score: 0.07171314741035857
```

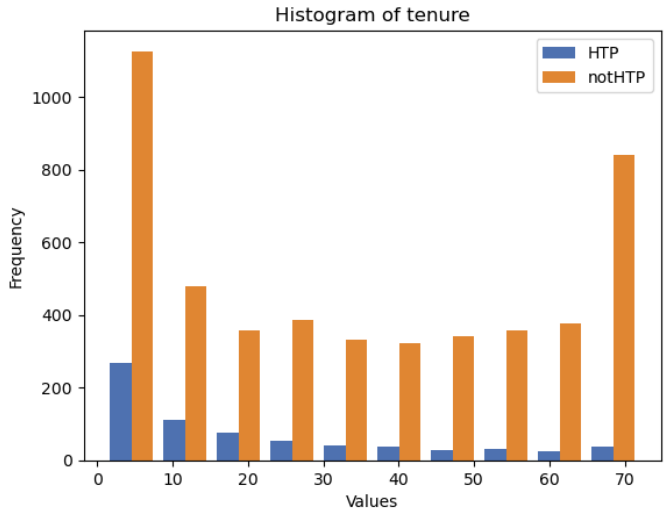
```
--------------------Gradient Boost--------
                              Importance
tenure                          0.222747
InternetService_Fiber optic     0.191900
TotalCharges                    0.158172
Contract_Two year               0.117015
MonthlyCharges                  0.112355
Training:
Confusion Matrix:
[[4919    0]
 [ 705    1]]

Accuracy: 0.8746666666666667
True-Positve Rate: 0.00141643059490085
F1 score: 0.002828854314002829
Testing:
Confusion Matrix:
[[1033    0]
 [ 373    1]]

Accuracy: 0.7348969438521677
True-Positve Rate: 0.00267379679144385
F1 score: 0.005333333333333333
```

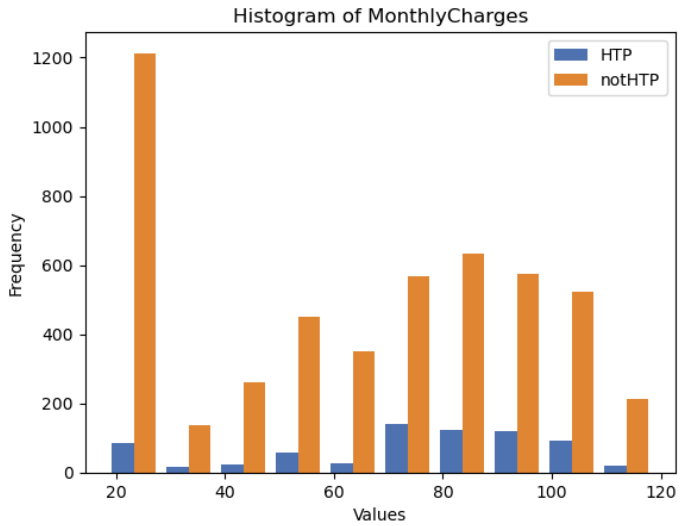
```
-------------------Regular Tree--------------------
                              Importance
TotalCharges                    0.289454
MonthlyCharges                  0.241974
tenure                          0.121496
gender_Male                     0.035319
InternetService_Fiber optic     0.029030
Training:
Confusion Matrix:
[[4918    1]
 [  11  695]]

Accuracy: 0.9978666666666667
True-Positve Rate: 0.9844192634560907
F1 score: 0.9914407988587732
Testomg:
Confusion Matrix:
[[912 121]
 [311  63]]

Accuracy: 0.6929637526652452
True-Positve Rate: 0.16844919786096257
F1 score: 0.22580645161290322
```

```
--------------------Random Forest--------------
                              Importance
TotalCharges                    0.217332
MonthlyCharges                  0.212104
tenure                          0.160666
gender_Male                     0.035558
PaperlessBilling_Yes            0.030327
Training:
Confusion Matrix:
[[4915    4]
 [   8  698]]

Accuracy: 0.9978666666666667
True-Positve Rate: 0.9886685552407932
F1 score: 0.9914772727272727
Testing:
Confusion Matrix:
[[1019   14]
 [ 363   11]]

Accuracy: 0.7320540156361052
True-Positve Rate: 0.029411764705882353
F1 score: 0.05513784461152882
```

3. Use most important variables from tree as proxy to determine which are HTP

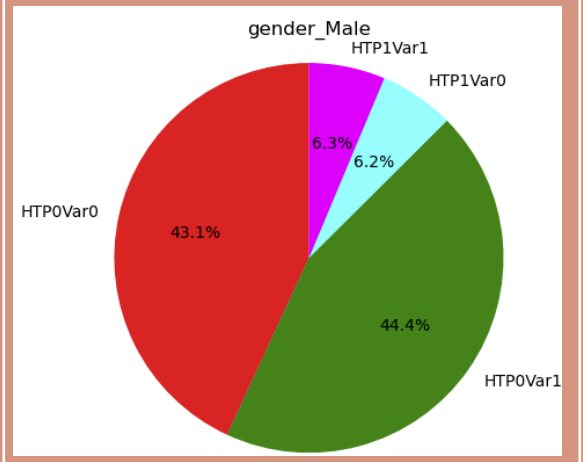


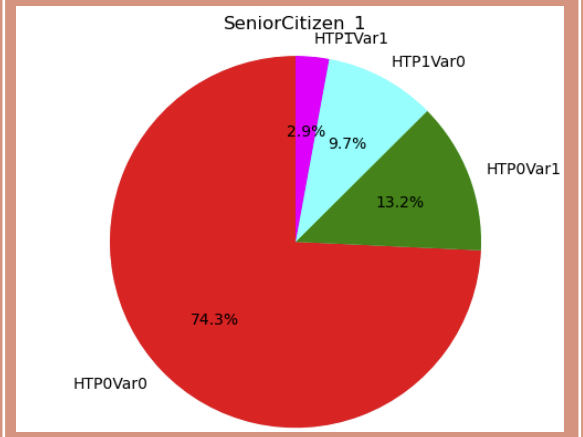


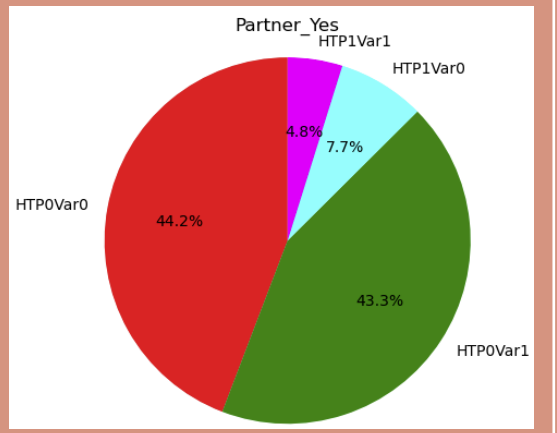


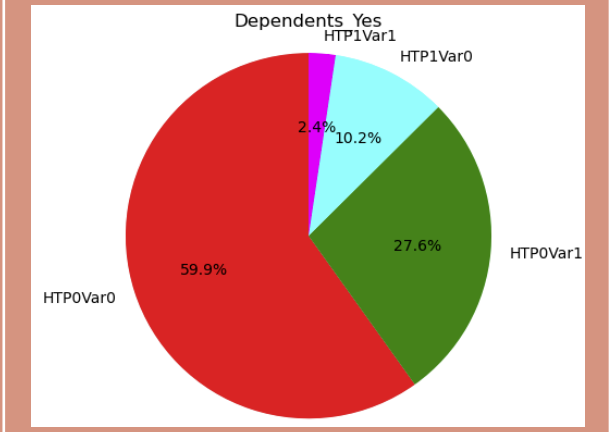
Histograms of Numerical variables

HTP

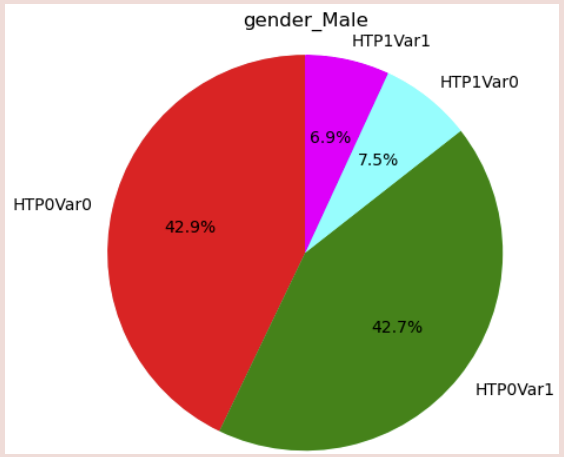

gender_Male
HTP1Var1
HTP1Var0
HTP0Var0
HTP0Var1
6.3%
6.2%
43.1%
44.4%

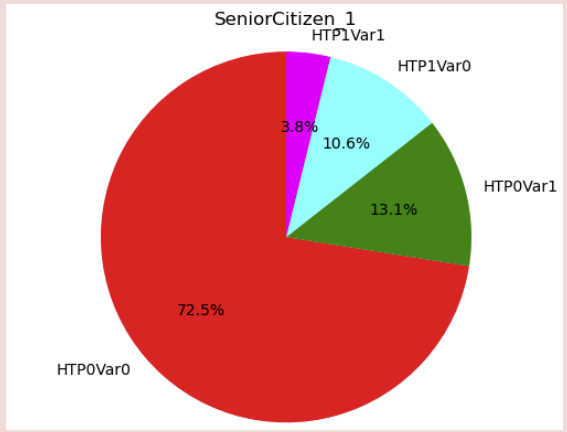

SeniorCitizen_1
HTP1Var1
HTP1Var0
HTP0Var1
HTP0Var0
2.9%
9.7%
13.2%
74.3%

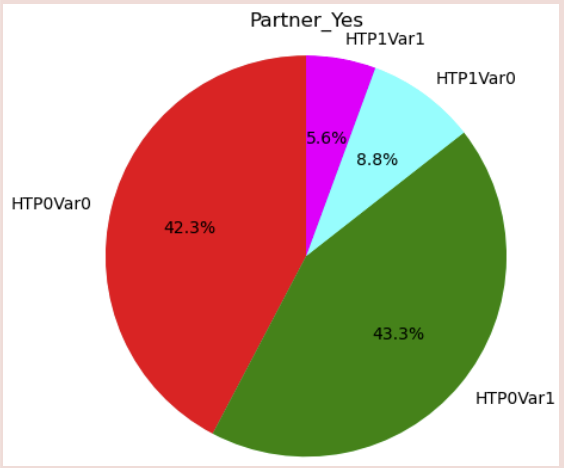

Partner_Yes
HTP1Var1
HTP1Var0
HTP0Var0
HTP0Var1
4.8%
7.7%
44.2%
43.3%

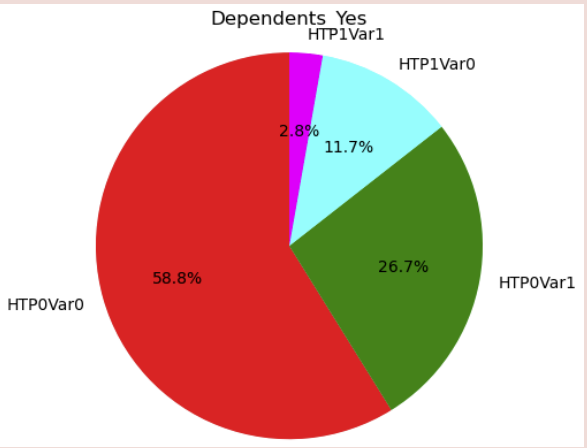

Dependents_Yes
HTP1Var1
HTP1Var0
HTP0Var0
HTP0Var1
2.4%
10.2%
59.9%
27.6%

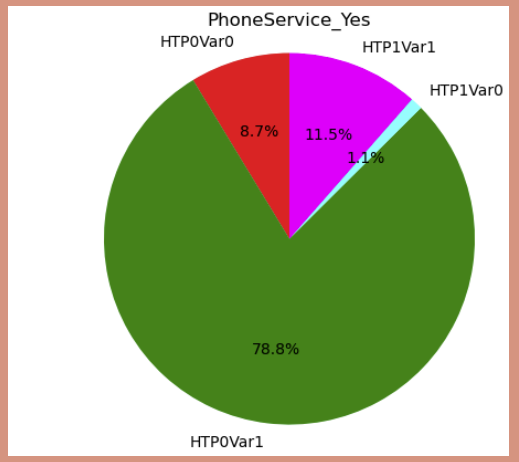
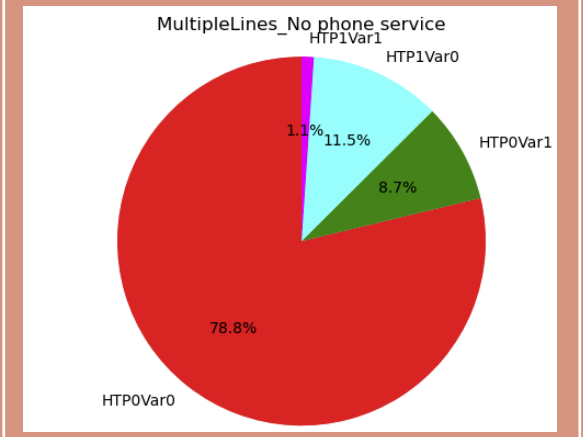
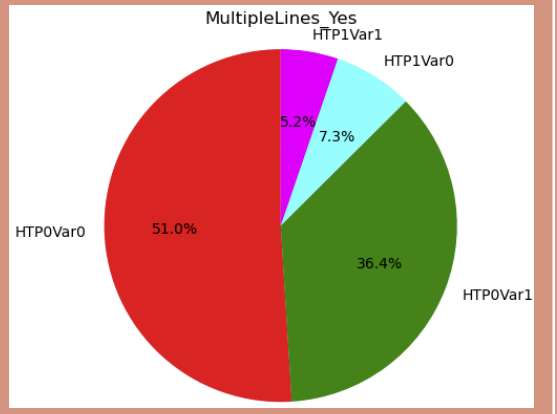
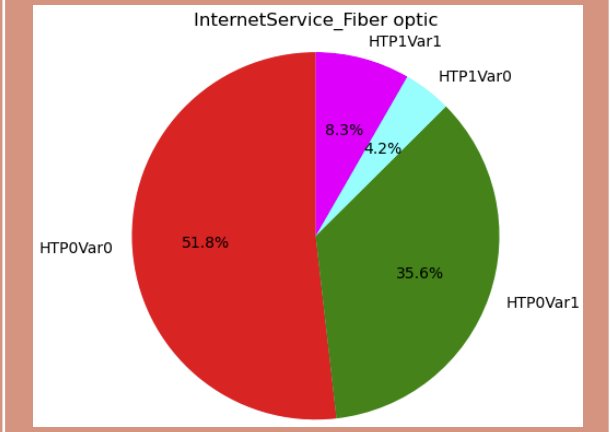
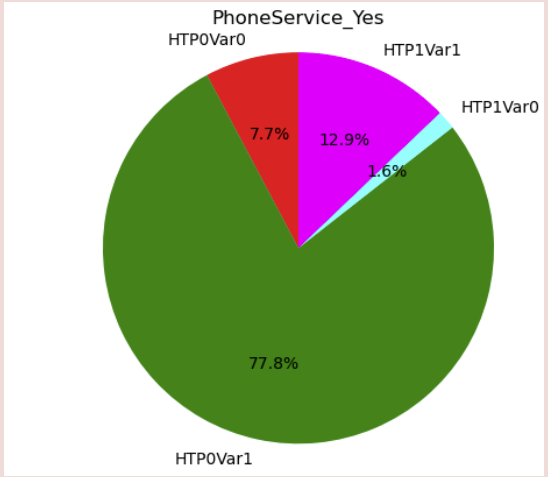
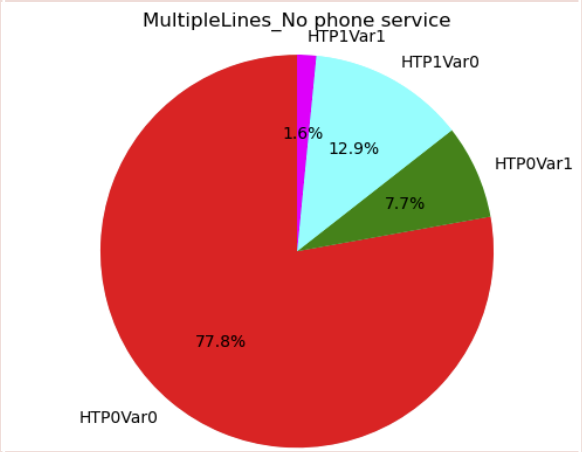
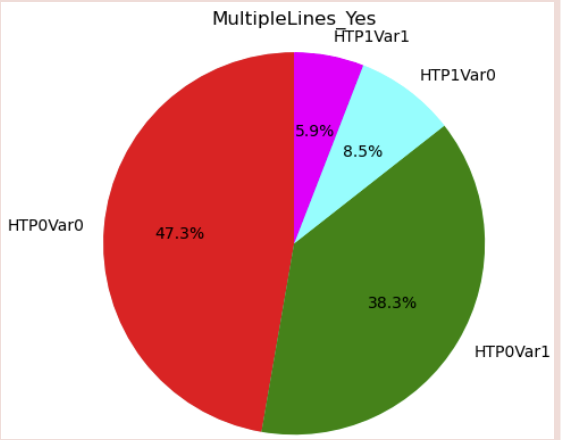
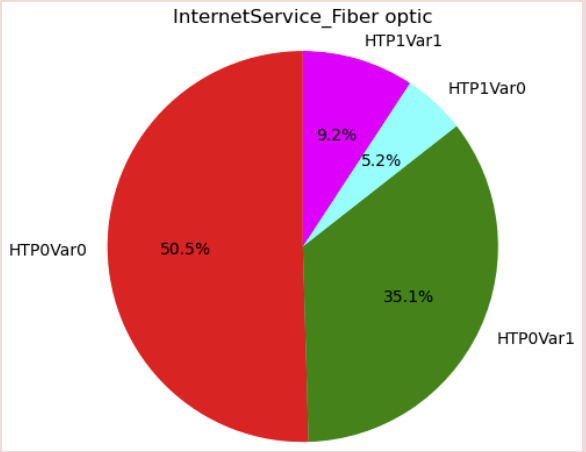
notHTP

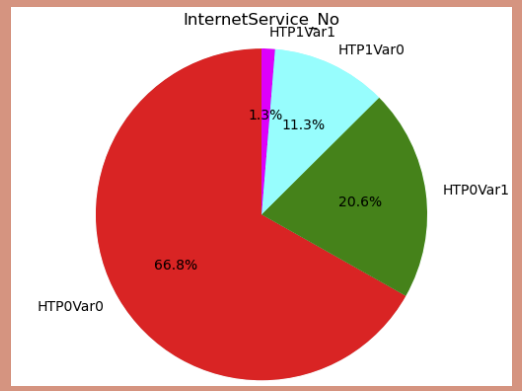
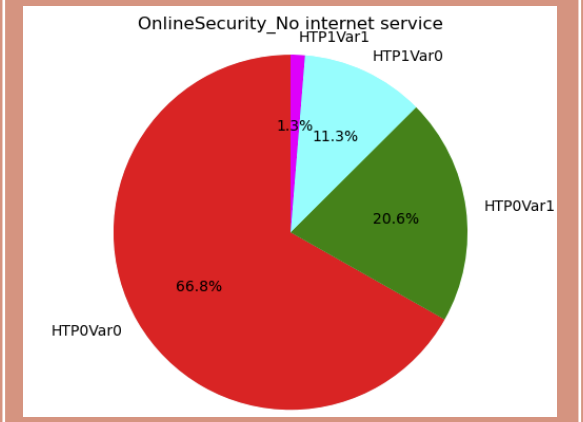
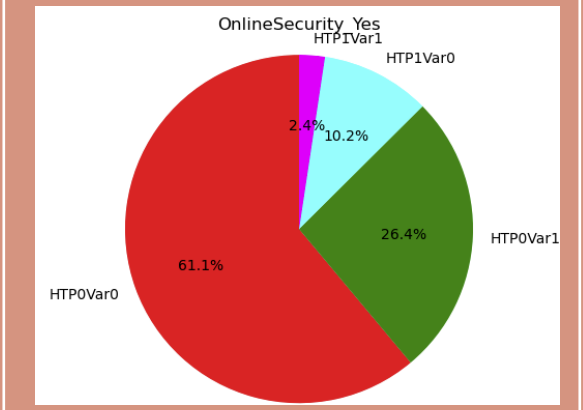
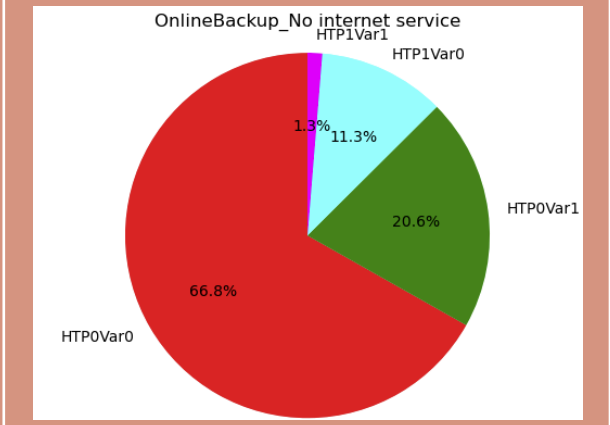
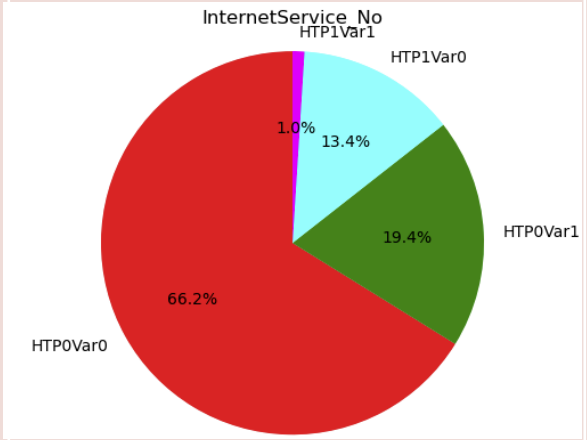
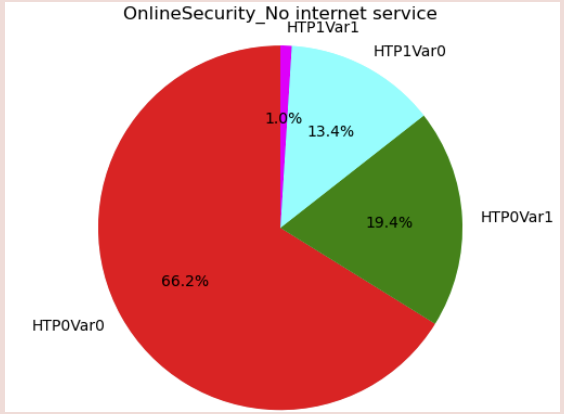
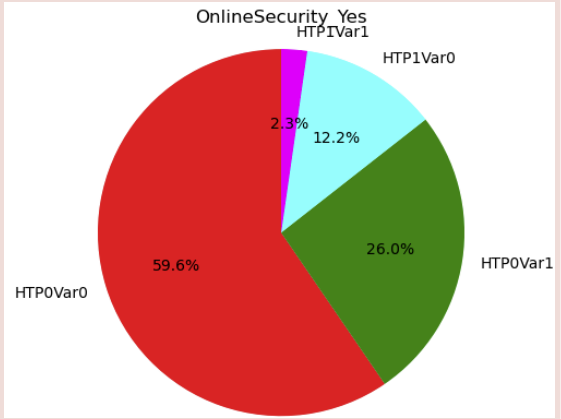
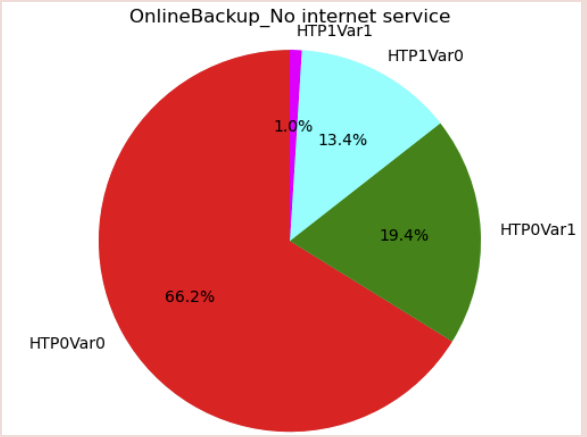

gender_Male
HTP1Var1
HTP1Var0
HTP0Var0
HTP0Var1
6.9%
7.5%
42.9%
42.7%


SeniorCitizen_1
HTP1Var1
HTP1Var0
HTP0Var1
HTP0Var0
3.8%
10.6%
13.1%
72.5%

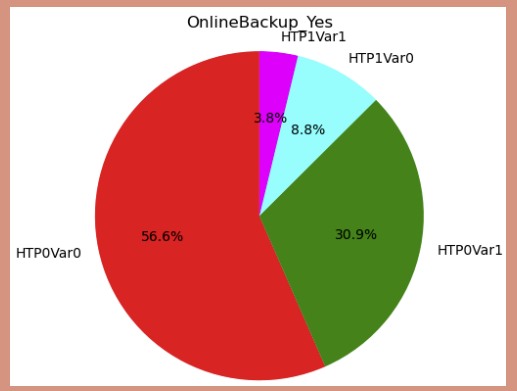

Partner_Yes
HTP1Var1
HTP1Var0
HTP0Var0
HTP0Var1
5.6%
8.8%
42.3%
43.3%


Dependents_Yes
HTP1Var1
HTP1Var0
HTP0Var0
HTP0Var1
2.8%
11.7%
58.8%
26.7%

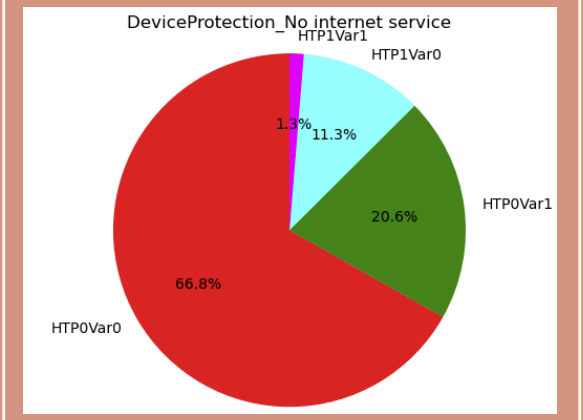
HTP	Var0 = HTP0	Var1 = HTP0		
				
notHTP				

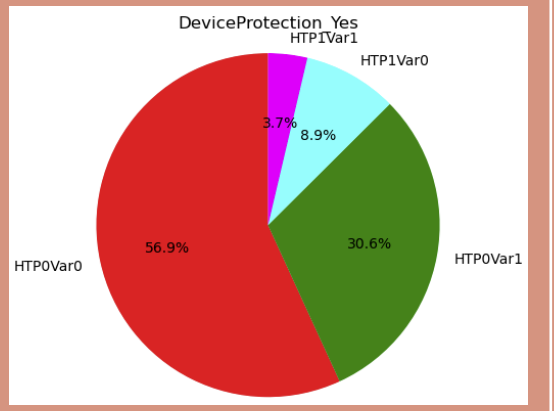
HTP	Var1 = HTP0	Var1 = HTP0		Var1 = HTP0
				
notHTP				

HTP

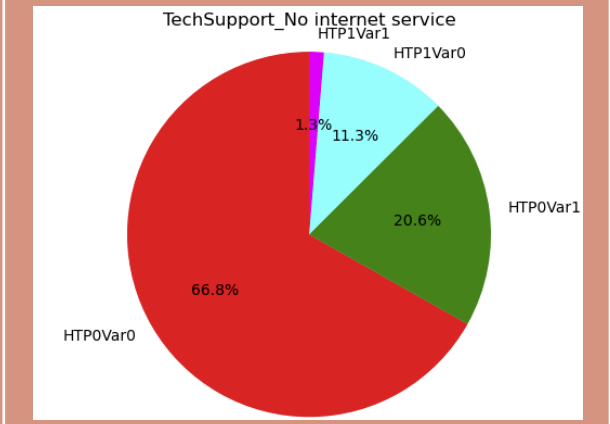


Var1 = HTP0

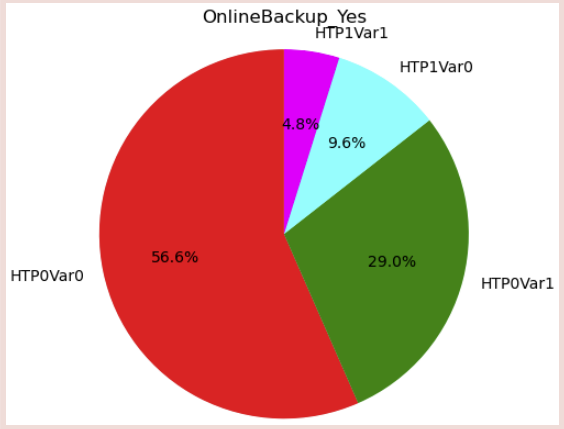


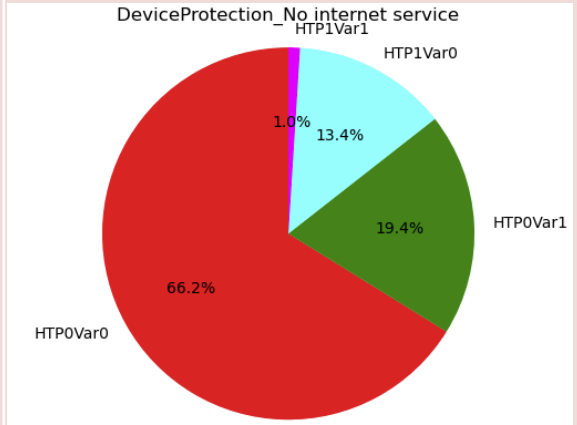


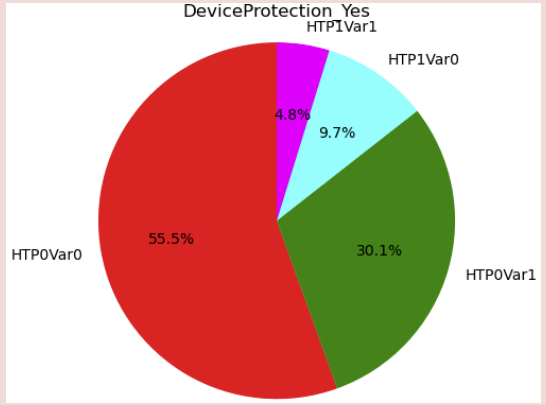
Var1 = HTP0

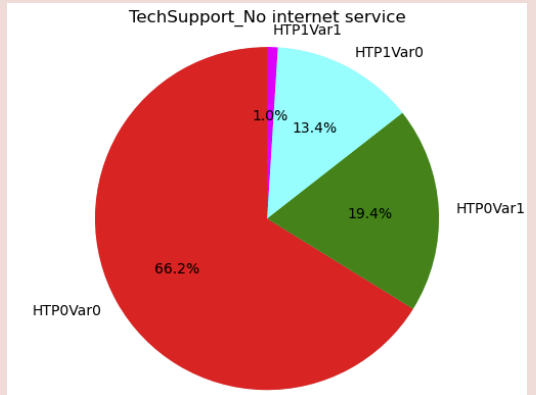


notHTP

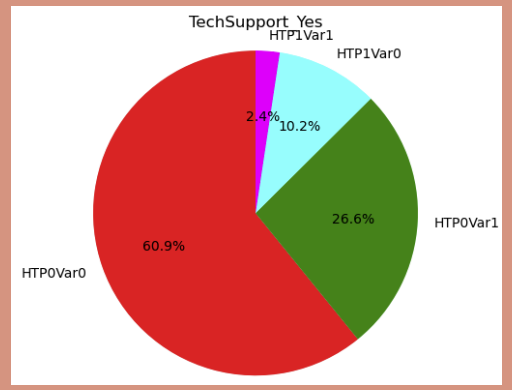




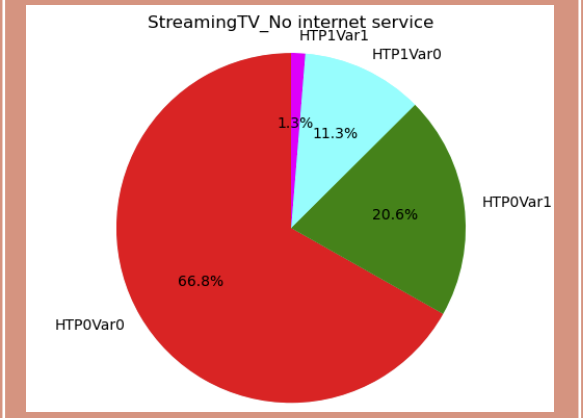


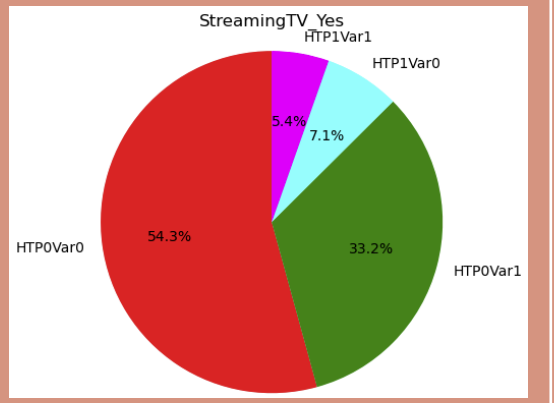


HTP

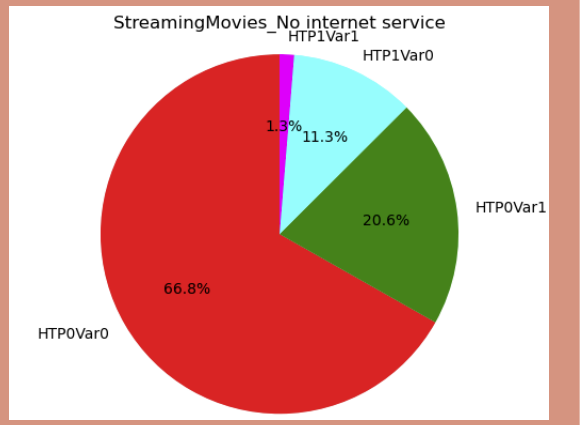


Var1 = HTP0

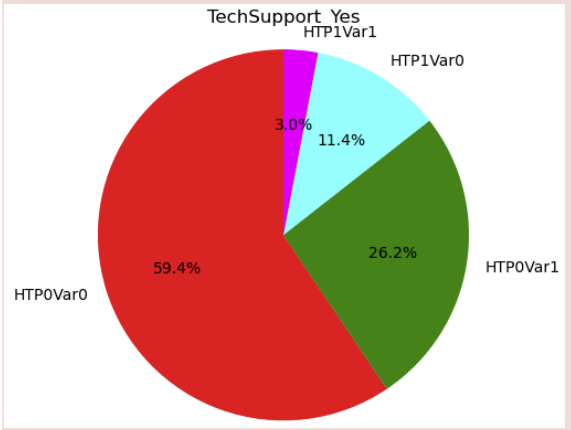


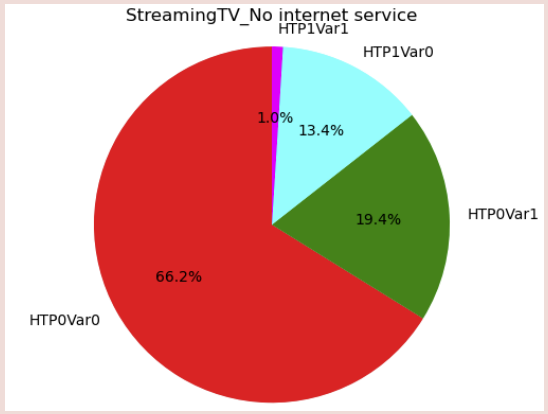


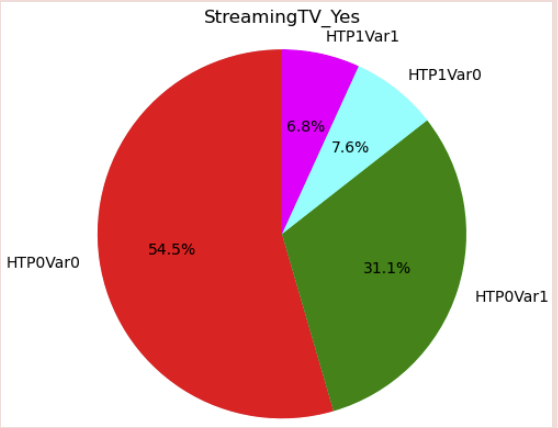
Var1 = HTP0

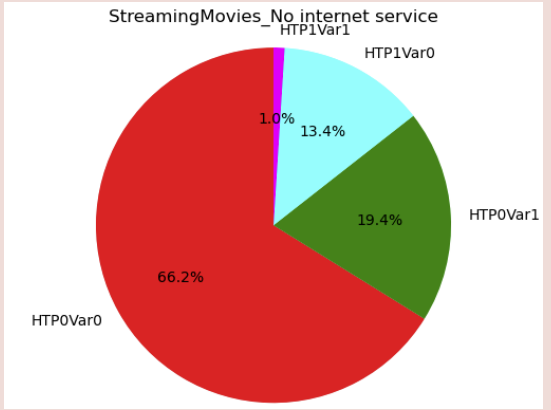


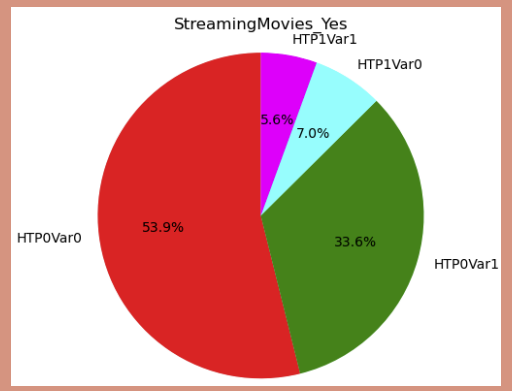
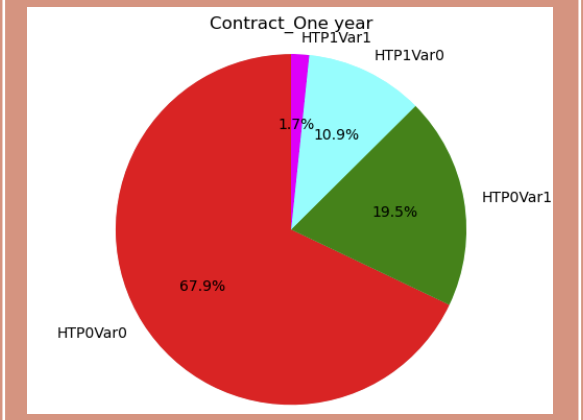
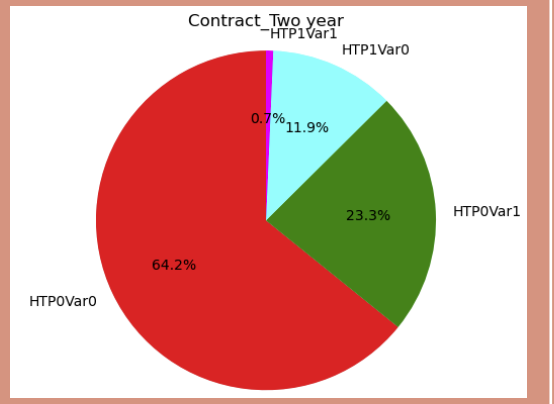
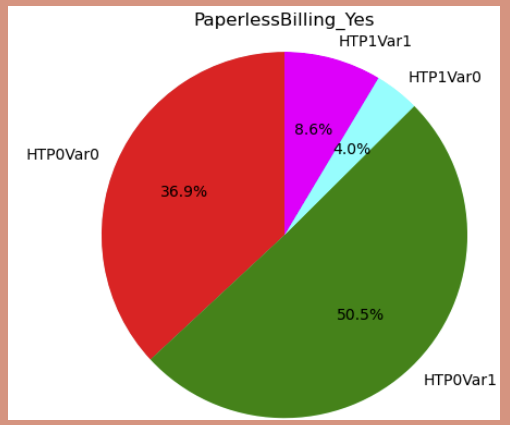
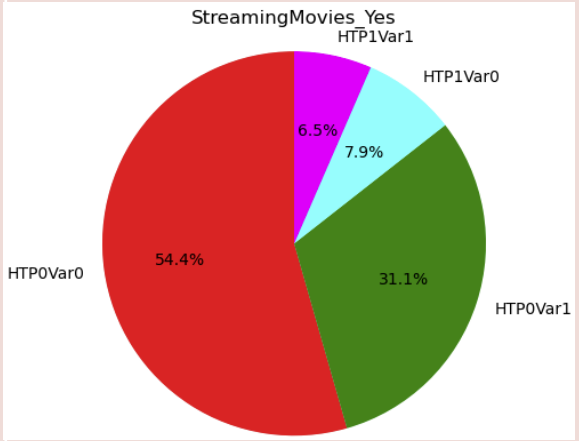
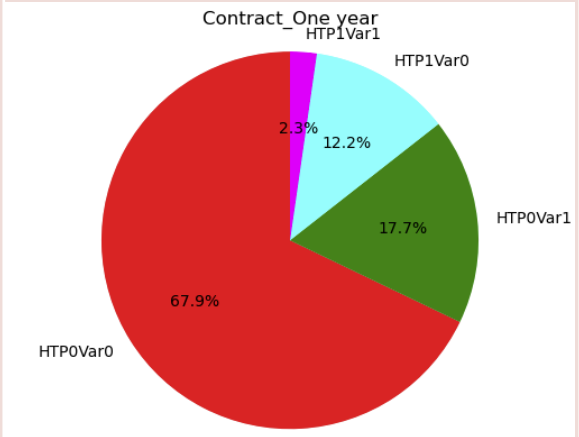
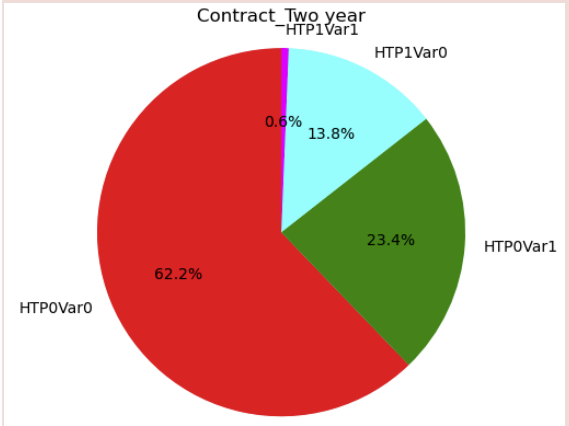
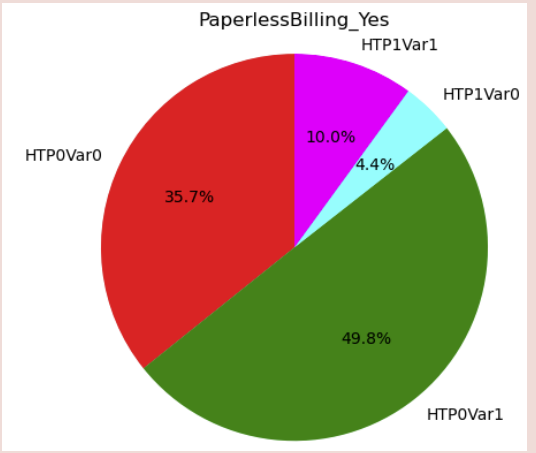
notHTP



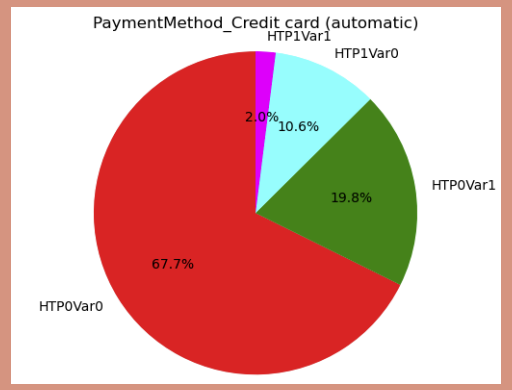


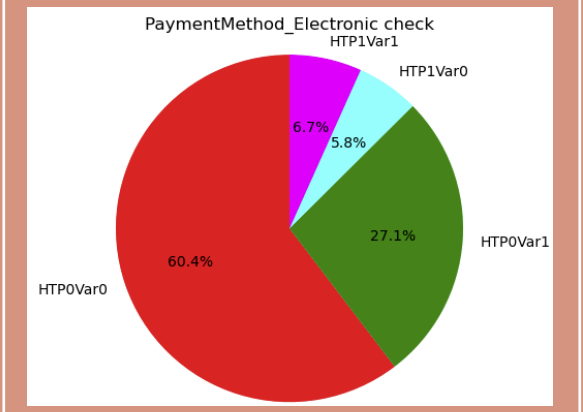


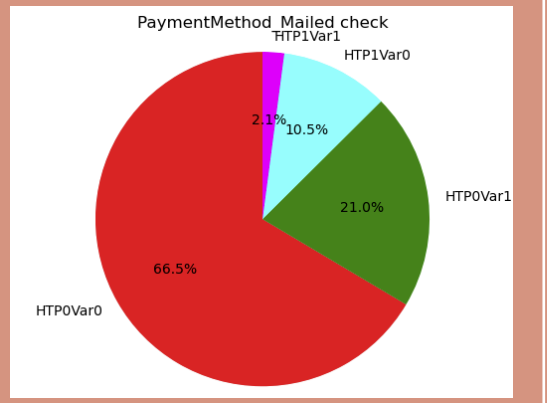


HTP		**Var1 = HTP0** 	**Var1 = HTP0** 	
notHTP				

HTP

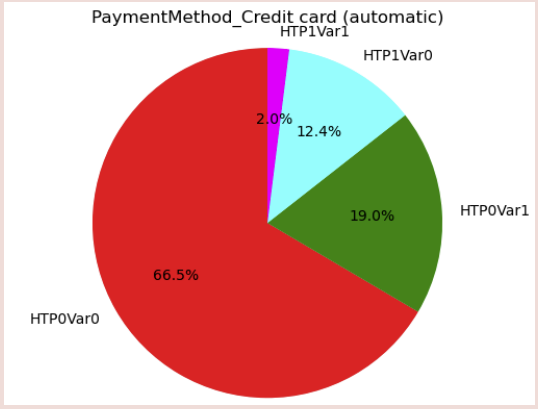


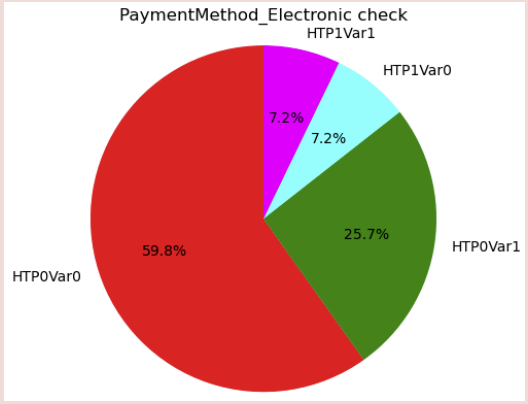


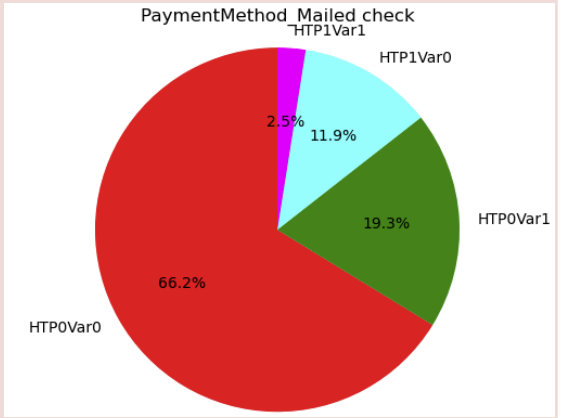


Most useful pie charts result in "Var1 = HTP0". This might not be enough to properly distinguish the two classes

notHTP







Performance

```
DF_HTP['predHTP'] = 1

DF_HTP.loc[DF_HTP['PhoneService_Yes'] == 0, 'predHTP'] = 0
DF_HTP.loc[DF_HTP['MultipleLines_No phone service'] == 1, 'predHTP'] = 0
DF_HTP.loc[DF_HTP['InternetService_No'] == 1, 'predHTP'] = 0
DF_HTP.loc[DF_HTP['OnlineSecurity_No internet service'] == 1, 'predHTP'] = 0
DF_HTP.loc[DF_HTP['OnlineBackup_No internet service'] == 1, 'predHTP'] = 0
DF_HTP.loc[DF_HTP['DeviceProtection_No internet service'] == 1, 'predHTP'] = 0
DF_HTP.loc[DF_HTP['TechSupport_No internet service'] == 1, 'predHTP'] = 0
DF_HTP.loc[DF_HTP['StreamingTV_No internet service'] == 1, 'predHTP'] = 0
DF_HTP.loc[DF_HTP['StreamingMovies_No internet service'] == 1, 'predHTP'] = 0
DF_HTP.loc[DF_HTP['Contract_One year'] == 1, 'predHTP'] = 0
DF_HTP.loc[DF_HTP['Contract_Two year'] == 1, 'predHTP'] = 0

print(len(DF_HTP[DF_HTP['predHTP'] == 1]))
print(len(DF_HTP[DF_HTP['predHTP'] == 0]))

evaluate(DF_HTP['HTP'], DF_HTP['predHTP'])
```

```
2358
3267
Confusion Matrix:
[[3267 1897]
 [   0  461]]

Accuracy: 0.6627555555555555
True-Positve Rate: 1.0
F1 score: 0.32706633557999293
```

HTP →

```
DF_HTP_Test['predHTP'] = 1

DF_HTP_Test.loc[DF_HTP_Test['PhoneService_Yes'] == 0, 'predHTP'] = 0
DF_HTP_Test.loc[DF_HTP_Test['MultipleLines_No phone service'] == 1, 'predHTP'] = 0
DF_HTP_Test.loc[DF_HTP_Test['InternetService_No'] == 1, 'predHTP'] = 0
DF_HTP_Test.loc[DF_HTP_Test['OnlineSecurity_No internet service'] == 1, 'predHTP'] = 0
DF_HTP_Test.loc[DF_HTP_Test['OnlineBackup_No internet service'] == 1, 'predHTP'] = 0
DF_HTP_Test.loc[DF_HTP_Test['DeviceProtection_No internet service'] == 1, 'predHTP'] = 0
DF_HTP_Test.loc[DF_HTP_Test['TechSupport_No internet service'] == 1, 'predHTP'] = 0
DF_HTP_Test.loc[DF_HTP_Test['StreamingTV_No internet service'] == 1, 'predHTP'] = 0
DF_HTP_Test.loc[DF_HTP_Test['StreamingMovies_No internet service'] == 1, 'predHTP'] = 0
DF_HTP_Test.loc[DF_HTP_Test['Contract_One year'] == 1, 'predHTP'] = 0
DF_HTP_Test.loc[DF_HTP_Test['Contract_Two year'] == 1, 'predHTP'] = 0

print(len(DF_HTP_Test[DF_HTP_Test['predHTP'] == 1]))
print(len(DF_HTP_Test[DF_HTP_Test['predHTP'] == 0]))

evaluate(DF_HTP_Test['HTP'], DF_HTP_Test['predHTP'])
```

```
617
790
Confusion Matrix:
[[720 484]
 [ 70 133]]

Accuracy: 0.6062544420753376
True-Positve Rate: 0.6551724137931034
F1 score: 0.32439024390243903
```

CH →

```
Total Confusion Matrix:
[[778 255]
 [290  84]]

Accuracy: 0.6126510305614783
True-Positve Rate: 0.22459893048128343
F1 score: 0.2356241234221599
```

Performance using HTP NN

```
MakeNN(DF_HTP[xColumns], DF_HTP['HTP'], DF_HTP_Test[xColumns], DF_HTP_Test['HTP'])
```

```
176/176 [==============================] - 0s 825us/step
44/44 [==============================] - 0s 812us/step
Confusion Matrix:
[[846 358]
 [ 84 119]]

Accuracy: 0.6858564321250888
True-Positve Rate: 0.5862068965517241
F1 score: 0.35
```

```
Total Confusion Matrix:
[[897 136]
 [246 128]]

Accuracy: 0.728500355366027
True-Positve Rate: 0.3422459893048128
F1 score: 0.4012539184952978
```

Week 10

To-Do

1. Compare customers that all (regular, notHTP, Separate HTP/notHTP, pie charts, perfect, log model) missed

2. Make arguments

- NN time need is too much compared to performance

- Separating HTP and notHTP leads to good performance, but its hard to separate

1. Compare misses

Class	NNPred	notHTPPred	PiePred	PerfectPred	HTPNN	numMissed
0	0	0	0.0	0.0	0.0	0.0
0	0	0	0.0	0.0	0.0	0.0
0	0	0	0.0	0.0	0.0	0.0
0	0	0	0.0	0.0	0.0	0.0
0	0	0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...
0	0	0	0.0	0.0	1.0	5.0
0	1	0	0.0	0.0	0.0	5.0
0	0	0	0.0	0.0	0.0	6.0
0	0	0	0.0	0.0	0.0	6.0
0	0	0	0.0	0.0	0.0	6.0

```
for i in range(0,7):
    print("Number of sample units missed", i,
    print("Number of sample units also HTP:",
    print("_____
```

```
Number of sample units missed 0 times: 248
Number of sample units also HTP: 0

_____
Number of sample units missed 1 times: 716
Number of sample units also HTP: 0

_____
Number of sample units missed 2 times: 109
Number of sample units also HTP: 0

_____
Number of sample units missed 3 times: 102
Number of sample units also HTP: 43

_____
Number of sample units missed 4 times: 198
Number of sample units also HTP: 155

_____
Number of sample units missed 5 times: 31
Number of sample units also HTP: 5

_____
Number of sample units missed 6 times: 3
Number of sample units also HTP: 0

_____
```

1. Compare misses

Churn_Yes	Prob	Class	NNPred	notHTPPred	PiePred	PerfectPred	HTPNN	numMissed	HTP
0	0.775030	1	1	1	1.0	0.0	0.0	4.0	1
0	0.647251	1	1	1	1.0	0.0	0.0	4.0	1
1	0.232114	0	0	0	0.0	1.0	1.0	4.0	1
0	0.437510	1	1	1	0.0	1.0	0.0	4.0	0
0	0.619405	1	1	1	1.0	0.0	0.0	4.0	1
...	...	...	...	...	...	...	...	...	...
0	0.610872	1	1	1	1.0	0.0	0.0	4.0	1
1	0.178144	0	0	0	0.0	1.0	1.0	4.0	1
0	0.634613	1	1	1	1.0	0.0	0.0	4.0	1
1	0.150323	0	0	0	0.0	1.0	1.0	4.0	1
0	0.708626	1	1	1	1.0	0.0	0.0	4.0	1

Conclusion 1

For simpler/smaller binary classification, the time needed for neural network is not worth the marginal improvement.

```
--------------------Log Model------------
Confusion Matrix:
[[833 200]
 [111 263]]

True Positive: 263
True Negative: 833
False Positive: 200
False Negative: 111

Accuracy: 0.7789623312011372
True-Positve Rate: 0.7032085561497327
F1 score: 0.6284348864994026
Execution time: 0.2592 seconds
```

```
--------------------Neural Network--------------------
176/176 [==============================] - 0s 864us/step
44/44 [==============================] - 0s 579us/step
Confusion Matrix:
[[802 231]
 [136 238]]

True Positive: 238
True Negative: 802
False Positive: 231
False Negative: 136

Accuracy: 0.7391613361762616
True-Positve Rate: 0.6363636363636364
F1 score: 0.5646500593119811
Execution time: 44.3693 seconds
```

```
--------------------Gradient Boost--------------------
Testing:
Confusion Matrix:
[[936  97]
 [195 179]]

True Positive: 179
True Negative: 936
False Positive: 97
False Negative: 195

Accuracy: 0.7924662402274343
True-Positve Rate: 0.4786096256684492
F1 score: 0.5507692307692309
Execution time: 0.6151 seconds
```

Conclusion 2

Separating HTP and notHTP leads to good performance, but its hard to separate

```
Confusion Matrix:
[[131   0]
 [  0  72]]

True Positive: 72
True Negative: 131
False Positive: 0
False Negative: 0

Accuracy: 1.0
True-Positve Rate: 1.0
F1 score: 1.0
```

```
Confusion Matrix:
[[873  29]
 [ 63 239]]

True Positive: 239
True Negative: 873
False Positive: 29
False Negative: 63

Accuracy: 0.9235880398671097
True-Positve Rate: 0.7913907284768212
F1 score: 0.8385964912280702
```

```
Total Confusion Matrix:
[[1004   29]
 [  63  311]]

Accuracy: 0.9346126510305615
True-Positve Rate: 0.8315508021390374
F1 score: 0.8711484593837535
```

```
MakeNN(DF_HTP[xColumns], DF_HTP['HTP'], DF_HTP_Test[xColumns], DF_HTP_Test['HTP'])

176/176 [==============================] - 0s 825us/step
44/44 [==============================] - 0s 812us/step
Confusion Matrix:
[[846 358]
 [ 84 119]]

Accuracy: 0.6858564321250888
True-Positve Rate: 0.5862068965517241
F1 score: 0.35
```

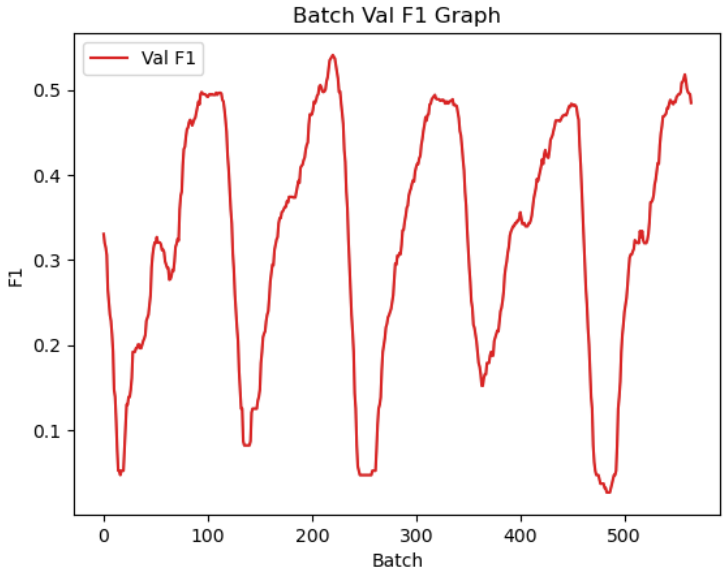
Conclusion 3

Prioritizing certain sample units does not guarantee better performance.

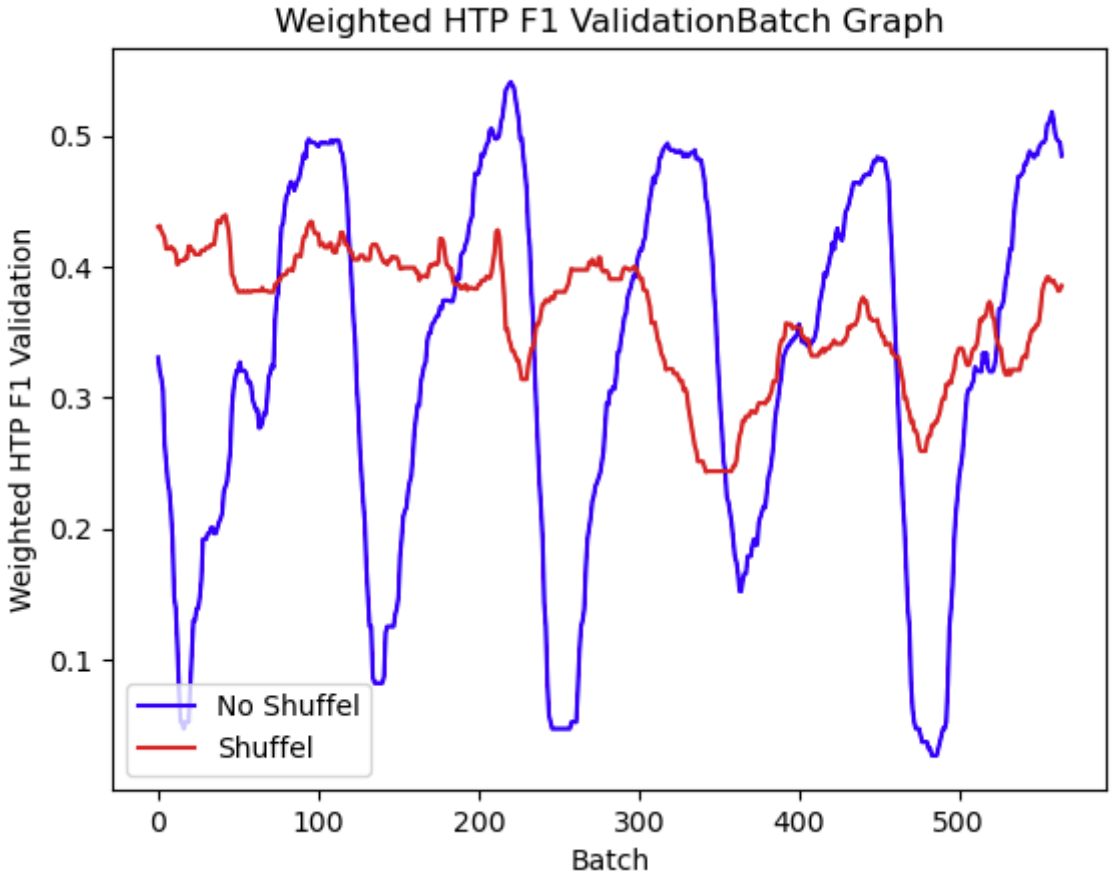
Slide 93 and 94

HTP(RWS)

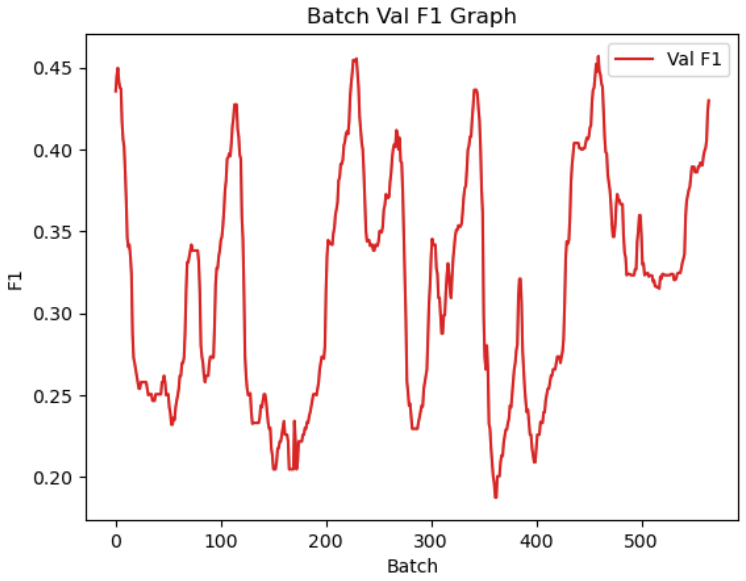
```
Best Threshold: 0.27
Accuracy: 0.7619047619047619
Recall: 0.679144385026738
F1: 0.6026097271648873
Confusion Matrix:
[[818 215]
 [120 254]]

True Positive: 254
True Negative: 818
False Positive: 215
False Negative: 120
```




```
Best Threshold: 0.21
Accuracy: 0.7540867093105899
Recall: 0.7058823529411765
F1: 0.6041189931350114
Confusion Matrix:
[[797 236]
 [110 264]]

True Positive: 264
True Negative: 797
False Positive: 236
False Negative: 110

Accuracy: 0.7540867093105899
True-Positve Rate: 0.7058823529411765
F1 score: 0.6041189931350114
Bootstrapped: 0.605710068055737
```



Save(RWS)




```
Best Threshold: 0.21
Accuracy: 0.7654584221748401
Recall: 0.6978609625668449
F1: 0.6126760563380282
Confusion Matrix:
[[816 217]
 [113 261]]

True Positive: 261
True Negative: 816
False Positive: 217
False Negative: 113

Accuracy: 0.7654584221748401
True-Positve Rate: 0.6978609625668449
F1 score: 0.6126760563380282
Bootstrapped: 0.6172493315578895
```