# Churn Prediction
# &
# XG Boost Model Parameters

by Alex Dance

# Purpose:

- Look at a Churn Model
- Explain the XG Boost parameters
- See how parameters can be changed

So your model does this

# Agenda:

**Part 1 - Research**

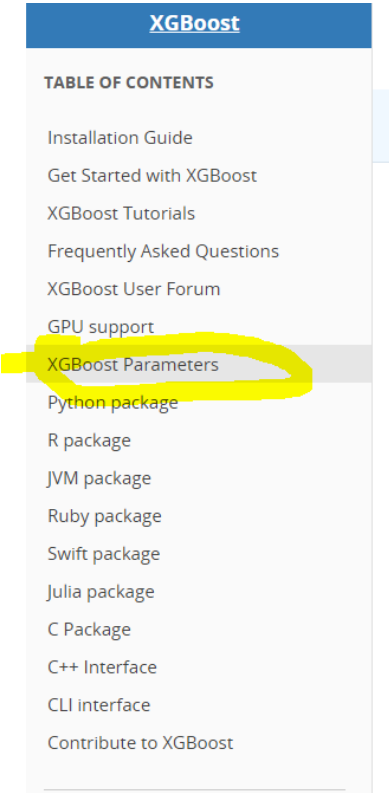
- Explain Parameter Changing
- Tips I liked

**Part 2 - Practice**

- My dataset
- EDA
- How to run parameters

and NOT this

# There are 3 main types of parameters




- **General parameters** relate to which booster we are using to do boosting, commonly tree or linear model
- **Booster parameters** depend on which booster you have chosen
- **Learning task parameters** decide on the learning scenario. For example, regression tasks may use different parameters with ranking tasks.
- **Command line parameters** relate to behavior of CLI version of XGBoost. (NOT relevant)

https://xgboost.readthedocs.io/en/latest/parameter.html

# There are a few ways of setting up parameters

XGBoost can use either a list of pairs or a dictionary to set parameters. For instance:

- Booster parameters

```
param = {'max_depth': 2, 'eta': 1, 'objective': 'binary:logistic'}
param['nthread'] = 4
param['eval_metric'] = 'auc'
```

- You can also specify multiple eval metrics:

```
param['eval_metric'] = ['auc', 'ams@0']

# alternatively:
# plst = param.items()
# plst += [('eval_metric', 'ams@0')]
```

- Specify validations set to watch performance

```
evallist = [(dtest, 'eval'), (dtrain, 'train')]
```

https://xgboost.readthedocs.io/en/latest/python/python_intro.html#setting-parameters

# Tip: a saved model can be exported and loaded

## Training¶

Training a model requires a parameter list and data set.

```
num_round = 10
bst = xgb.train(param, dtrain, num_round, evallist)
```

After training, the model can be saved.

```
bst.save_model('0001.model')
```

The model and its feature map can also be dumped to a text file.

```
# dump model
bst.dump_model('dump.raw.txt')
# dump model with feature map
bst.dump_model('dump.raw.txt', 'featmap.txt')
```

A saved model can be loaded as follows:

```
bst = xgb.Booster({'nthread': 4})  # init model
bst.load_model('model.bin')  # load data
```

Methods including update and boost from xgboost.Booster are designed for internal usage only. The wrapper function xgboost.train does some pre-configuration including setting up caches and some other parameters.

# Tip: Early stopping can still yield results

A model that has been trained or loaded can perform predictions on data sets.

```
# 7 entities, each contains 10 features
data = np.random.rand(7, 10)
dtest = xgb.DMatrix(data)
ypred = bst.predict(dtest)
```

If early stopping is enabled during training, you can get predictions from the best iteration with `bst.best_ntree_limit`:

```
ypred = bst.predict(dtest, ntree_limit=bst.best_ntree_limit)
```

If early stopping occurs, the model will have three additional fields: `bst.best_score`, `bst.best_iteration` and `bst.best_ntree_limit`. Note that `xgboost.train()` will return a model from the last iteration, not the best one.

This works with both metrics to minimize (RMSE, log loss, etc.) and to maximize (MAP, NDCG, AUC). Note that if you specify more than one evaluation metric the last one in `param['eval_metric']` is used for early stopping.

# Tip: Parameters running

From Vidhya

1.Verbosity of printing messages. Valid values are
0 (silent)
1 (warning)
2 (info)
3 (debug)

**nthread [default to maximum number of threads available if not set]**

1. This is used for parallel processing and number of cores in the system should be entered
2. If you wish to run on all cores, value should not be entered and algorithm will detect automatically

https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/

# Booster Parameters

From Vidhya
https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/

**eta [default=0.3]**
- Analogous to learning rate in GBM
- Makes the model more robust by shrinking the weights on each step
- Typical final values to be used: 0.01-0.2

**min_child_weight [default=1]**
- Defines the minimum sum of weights of all observations required in a child.
- This is similar to **min_child_leaf** in GBM but not exactly. This refers to min “sum of weights” of observations while GBM has min “number of observations
- Used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.
- Too high values can lead to under-fitting hence, it should be tuned using CV.

**max_depth [default=6]**
- The maximum depth of a tree, same as GBM.
- Used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.
- Should be tuned using CV.
- Typical values: 3-10

**max_leaf_nodes**
- The maximum number of terminal nodes or leaves in a tree.
- Can be defined in place of max_depth. Since binary trees are created, a depth of ‘n’ would produce a maximum of 2^n leaves.
- If this is defined, GBM will ignore max_depth.

**gamma [default=0]**
- A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split.
- Makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.

**max_delta_step [default=0]**
- In maximum delta step we allow each tree’s weight estimation to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative.
- Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced.
- This is generally not used but you can explore further if you wish.

**subsample [default=1]**
- Same as the subsample of GBM. Denotes the fraction of observations to be randomly samples for each tree.
- Lower values make the algorithm more conservative and prevents overfitting but too small values might lead to under-fitting.
- Typical values: 0.5-1

**colsample_bytree [default=1]**
- Similar to max_features in GBM. Denotes the fraction of columns to be randomly samples for each tree.
- Typical values: 0.5-1

**colsample_bylevel [default=1]**
- Denotes the subsample ratio of columns for each split, in each level.
- I don’t use this often because subsample and colsample_bytree will do the job for you. but you can explore further if you feel so.

**lambda [default=1]**
- L2 regularization term on weights (analogous to Ridge regression)
- This used to handle the regularization part of XGBoost. Though many data scientists don’t use it often, it should be explored to reduce overfitting.

**alpha [default=0]**
- L1 regularization term on weight (analogous to Lasso regression)
- Can be used in case of very high dimensionality so that the algorithm runs faster when implemented

**scale_pos_weight [default=1]**
- A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.

# Learning Task Parameters

From Vidhya

These parameters are used to define the optimization objective the metric to be calculated at each step.

**objective [default=reg:linear]**

This defines the loss function to be minimized. Mostly used values are:

- **binary:logistic** –logistic regression for binary classification, returns predicted probability (not class)
- **multi:softmax** –multiclass classification using the softmax objective, returns predicted class (not probabilities)
  - you also need to set an additional **num_class** (number of classes) parameter defining the number of unique classes
- **multi:softprob** –same as softmax, but returns predicted probability of each data point belonging to each class.

**eval_metric [ default according to objective ]**

The metric to be used for validation data.

The default values are rmse for regression and error for classification.

Typical values are:

- **rmse** – root mean square error
- **mae** – mean absolute error
- **logloss** – negative log-likelihood
- **error** – Binary classification error rate (0.5 threshold)
- **merror** – Multiclass classification error rate
- **mlogloss** – Multiclass logloss
- **auc:** Area under the curve

**seed [default=0]**

The random number seed.

Can be used for generating reproducible results and also for parameter tuning.

# General approach for parameter tuning

From Vidhya

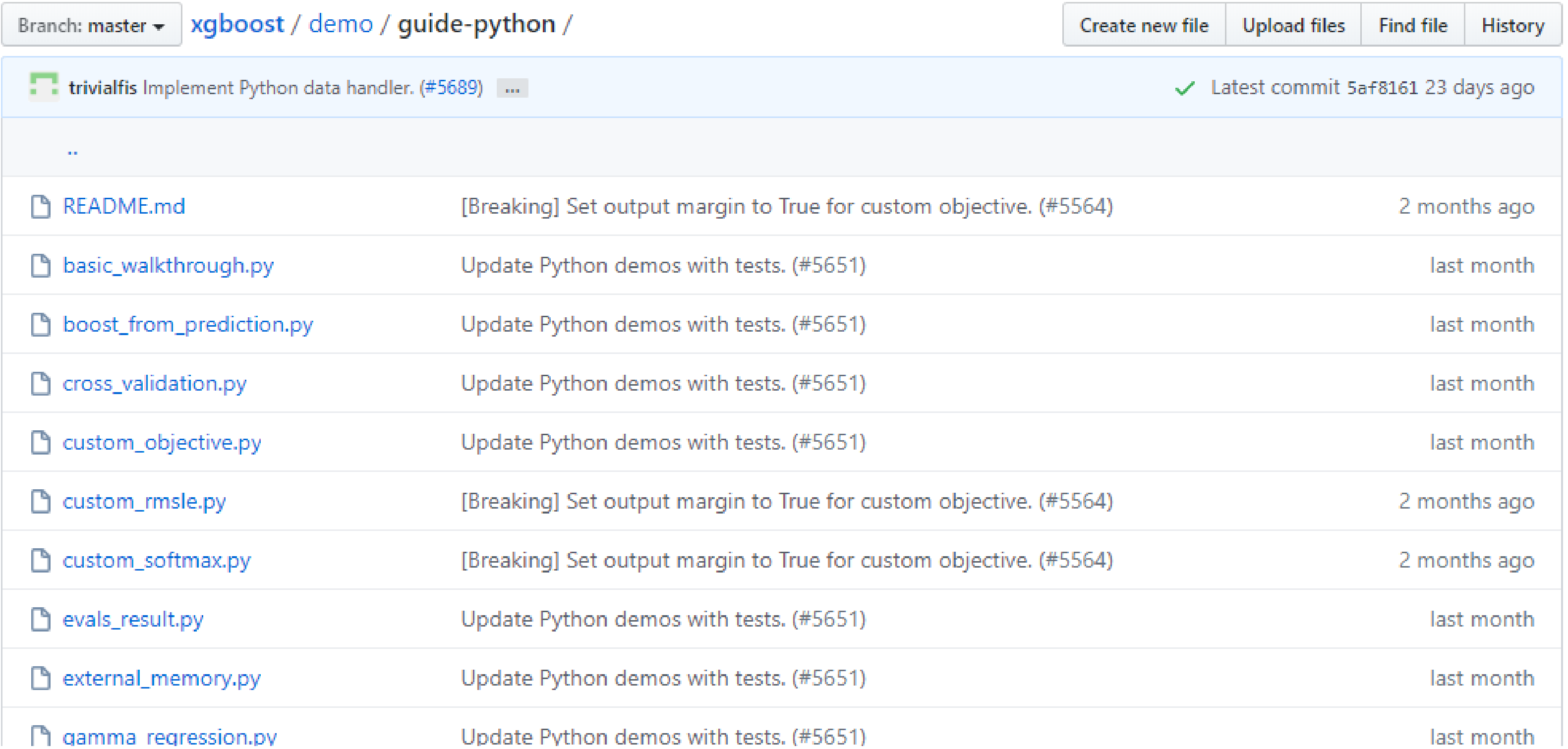
Choose a relatively **high learning rate**. Generally a learning rate of 0.1 works but somewhere between 0.05 to 0.3 should work for different problems.

Determine the **optimum number of trees for this learning rate**. XGBoost has a very useful function called as “cv” which performs cross-validation at each boosting iteration and thus returns the optimum number of trees required.

**Tune tree-specific parameters** ( max_depth, min_child_weight, gamma, subsample, colsample_bytree) for decided learning rate and number of trees. Note that we can choose different parameters to define a tree and I’ll take up an example here.

Tune **regularization parameters** (lambda, alpha) for xgboost which can help reduce model complexity and enhance performance.

# Tip: There is a good guide to XG Boost on GitHub


Branch: master ▾ xgboost / demo / guide-python /

Create new file | Upload files | Find file | History

trivialfis Implement Python data handler. (#5689) … ✓ Latest commit 5af8161 23 days ago

| | | |
|---|---|---|
| .. | | |
| README.md | [Breaking] Set output margin to True for custom objective. (#5564) | 2 months ago |
| basic_walkthrough.py | Update Python demos with tests. (#5651) | last month |
| boost_from_prediction.py | Update Python demos with tests. (#5651) | last month |
| cross_validation.py | Update Python demos with tests. (#5651) | last month |
| custom_objective.py | Update Python demos with tests. (#5651) | last month |
| custom_rmsle.py | [Breaking] Set output margin to True for custom objective. (#5564) | 2 months ago |
| custom_softmax.py | [Breaking] Set output margin to True for custom objective. (#5564) | 2 months ago |
| evals_result.py | Update Python demos with tests. (#5651) | last month |
| external_memory.py | Update Python demos with tests. (#5651) | last month |
| gamma_regression.py | Update Python demos with tests. (#5651) | last month |

https://github.com/dmlc/xgboost/tree/master/demo/guide-python

Now

my

code

# Churn and Acquisition are regular tasks

[https://www.kaggle.com/blastchar/telco-customer-churn](https://www.kaggle.com/blastchar/telco-customer-churn)


```
RangeIndex: 7043 entries, 0 to 7042
Data columns (total 21 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   customerID        7043 non-null   object
 1   gender            7043 non-null   object
 2   SeniorCitizen     7043 non-null   int64
 3   Partner           7043 non-null   object
 4   Dependents        7043 non-null   object
 5   tenure            7043 non-null   int64
 6   PhoneService      7043 non-null   object
 7   MultipleLines     7043 non-null   object
 8   InternetService   7043 non-null   object
 9   OnlineSecurity    7043 non-null   object
 10  OnlineBackup      7043 non-null   object
 11  DeviceProtection  7043 non-null   object
 12  TechSupport       7043 non-null   object
 13  StreamingTV       7043 non-null   object
 14  StreamingMovies   7043 non-null   object
 15  Contract          7043 non-null   object
 16  PaperlessBilling  7043 non-null   object
 17  PaymentMethod     7043 non-null   object
 18  MonthlyCharges    7043 non-null   float64
 19  TotalCharges      7043 non-null   object
 20  Churn             7043 non-null   object
```

# Cleaning the data

In [8]:

```
mapping = {'Yes':1, 'No':0}
df['Churn'] = df['Churn'].map(mapping)

```

In [9]:

```
df['Churn'].value_counts()
```

Out[9]:

```
0    5174
1    1869
Name: Churn, dtype: int64
```

]:

```
df['TotalCharges'] = pd.to_numeric(df['TotalCharges'],errors='coerce')
```

]:

```
type(df['MonthlyCharges'])
```

```
df = pd.get_dummies(data = df, columns = ['gender', 'Partner','Dependents','PhoneService','MultipleLines','InternetService',
```

```
df.head()
```

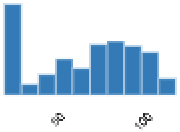
| customerID | SeniorCitizen | tenure | MonthlyCharges | TotalCharges | Churn | gender_Female | gender_Male | Partner_No | Partner_Yes | ... | PaperlessBilling_Yes |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 7590-VHVEG | 0 | 1 | 29.85 | 29.85 | 0 | 1 | 0 | 0 | 1 | ... | 1 |
| 5575-GNVDE | 0 | 34 | 56.95 | 1889.5 | 0 | 0 | 1 | 1 | 0 | ... | 0 |
| 3668-QPYBK | 0 | 2 | 53.85 | 108.15 | 1 | 0 | 1 | 1 | 0 | ... | 1 |

# Looked at the data

**MonthlyCharges**
Real number ($\mathbb{R}_{\geq 0}$)

| | |
|---|---|
| Distinct count | 1585 |
| Unique (%) | 22.5% |
| Missing | 0 |
| Missing (%) | 0.0% |
| Infinite | 0 |
| Infinite (%) | 0.0% |

| | |
|---|---|
| Mean | 64.76169246059 |
| Minimum | 18.25 |
| Maximum | 118.75 |
| Zeros | 0 |
| Zeros (%) | 0.0% |
| Memory size | 55.1 KiB |

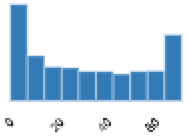


Toggle details

**tenure**
Real number ($\mathbb{R}_{\geq 0}$)

| | |
|---|---|
| Distinct count | 73 |
| Unique (%) | 1.0% |
| Missing | 0 |
| Missing (%) | 0.0% |
| Infinite | 0 |
| Infinite (%) | 0.0% |

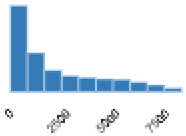
| | |
|---|---|
| Mean | 32.37114865824 |
| Minimum | 0 |
| Maximum | 72 |
| Zeros | 11 |
| Zeros (%) | 0.2% |
| Memory size | 55.1 KiB |



**TotalCharges**
Real number ($\mathbb{R}_{\geq 0}$)

| | |
|---|---|
| Distinct count | 6531 |
| Unique (%) | 92.7% |
| Missing | 0 |
| Missing (%) | 0.0% |
| Infinite | 0 |
| Infinite (%) | 0.0% |

| | |
|---|---|
| Mean | 2283.30044084 |
| Minimum | 18.8 |
| Maximum | 8684.8 |
| Zeros | 0 |
| Zeros (%) | 0.0% |
| Memory size | 55.1 KiB |



# Looking at Dependent Variables

```
df.groupby(["Churn"])["tenure"].mean()
```

```
Churn
0    37.569965
1    17.979133
Name: tenure, dtype: float64
```

```
df.groupby(["Churn"])["Partner_No"].mean()
```

```
Churn
0    0.471782
1    0.642055
Name: Partner_No, dtype: float64
```

```
df.groupby(["Churn"])["TotalCharges"].mean()
```

```
Churn
0    2555.344141
1    1531.796094
Name: TotalCharges, dtype: float64
```

More likely to churn if

Tenure was LOWER

Spend was Lower

# Ran Oversampling

```
train_b2, test_b2 = train_test_split(df, test_size=0.20, random_state=1)
```

```
target_count = train_b2.Churn.value_counts()
```

```
print(target_count)
```

```
0    4113
1    1521
Name: Churn, dtype: int64
```

```
count_class_1 =target_count[1]
count_class_0 = target_count[0]
```

In [71]:

```
df_class_0 = train_b2[train_b2['Churn'] == 0]
df_class_1 = train_b2[train_b2['Churn'] == 1]
```

In [72]:

```
# Random over-sampling
df_class_1_over = df_class_1.sample(count_class_0, replace=True)
df_train_over = pd.concat([df_class_0, df_class_1_over], axis=0)
```

In [74]:

```
df_train_over['Churn'].value_counts()
```

Out[74]:

```
1    4113
0    4113
Name: Churn, dtype: int64
```

In [75]:

```
# Get all valriables
```

In [76]:

```
y_test_bs = test_b2  ['Churn']
X_test_bs = test_b2.drop(["customerID","Churn"], axis = 1)

y_train_bs = df_train_over[['Churn']]
X_train_bs = df_train_over.drop(["customerID","Churn"], axis = 1)

```

# 4 Basic XG Boost  Models

Default Limited Features

```
XG_Results Results

XG_Results AUC Test  79.69%
[[965  96]
 [199 149]]
XG_Results accuracy_score_train  77.65%
XG_Results accuracy_score_test  79.06%
```

All – except  - top 3 features

```
XG_Results AUC Test  79.55%
[[878 183]
 [153 195]]
XG_Results accuracy_score_train  88.62%
XG_Results accuracy_score_test  76.15%
```

X_b = df.drop(["customerID","Churn”], axis = 1)

All With oversampling

```
xG train score 94.98%
XG test score 76.30%
```

All - Standard

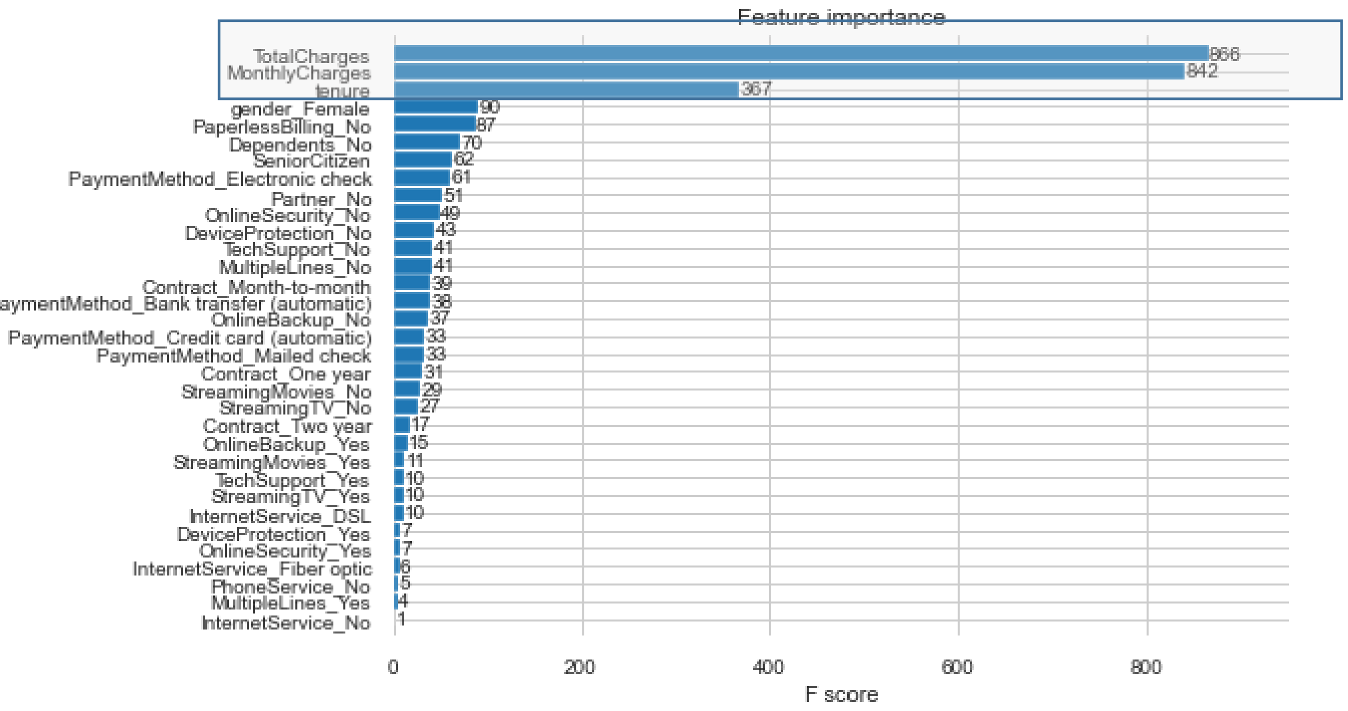
```
XG_Results Results

XG_Results AUC Test  83.74%
[[926 135]
 [147 201]]
XG_Results accuracy_score_train  93.81%
XG_Results accuracy_score_test  79.99%
```

Plus Logistic regression - limited

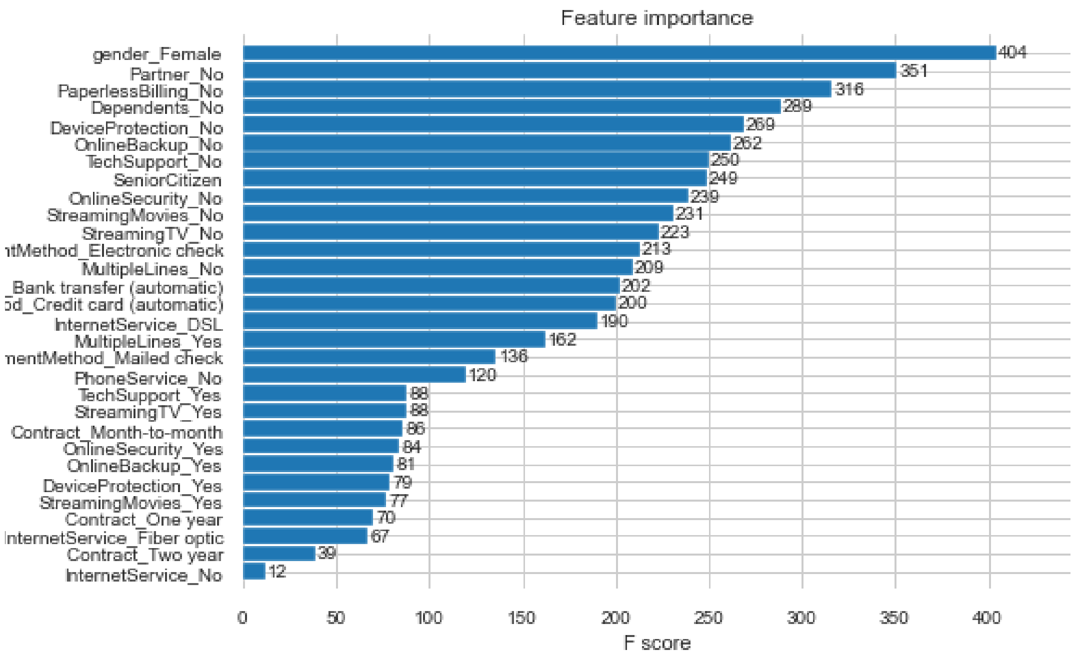
```
train score 77%
test score 79%
```

# Looked at the feature importance

Original




Updated




Could be Negatively or Positively Correlated

# Grid Search CV

```
from sklearn.model_selection import GridSearchCV
```

```
xgb_modecv = XGBClassifier(params = params)
```

```
test_params = {
 'max_depth':[4,8,12]
}
```

1

```
model = GridSearchCV(estimator = xgb_modecv,param_grid = test_params)
```

```
model.fit(X_train_b,y_train_b)
```

In [117]:

```
test_params = {
 'max_depth':[2,4,8,12]
}
```

:

```
print (model.best_params_)
```

```
{'max_depth': 2}
```

# Iterated over Function to check everything

Set up loop

Parameters Set initially

Numerical increased over iteration

Categorical Changed manually

Ran Model

See results

7 Loops

12 Variables

Over 50 different tests

# Was able to change multiple parameters

Can comment out all but 1 variable

Start

```
LearningRate = 0.1
MaxDepth = 1
Alpha = 1
Grow_Policy = 'lossguide'
ColSampleByTree = 0.1
MaximumDepth = 1 #'alpha'
Colsample_bytree = 0.3
Colsample_bylevel = 0.3
Colsample_bynode = 0.3
TreeMethod = 'auto'
Process_type = 'default'
```

Changes

```
# Every time we go through change values
LearningRate += 0.1
MaxDepth +=1
ColSampleByTree +=0.1
Alpha +=1
Colsample_bytree = Colsample_bytree +0.1
Colsample_bylevel = Colsample_bylevel +0.1
Colsample_bynode = Colsample_bynode +0.1
if i == 1:
    MaximumDepth = 1
    TreeMethod = 'exact'

if i > 1:
    Grow_Policy = 'depthwise'
    MaximumDepth += 1
    TreeMethod = 'approx'
    Updater = 'prune'
```

```
# Every time we go through change values
LearningRate += 0.1
#MaxDepth +=1
#ColSampleByTree +=0.1
#Alpha +=1
#Colsample_bytree = Colsample_bytree +0.1
#Colsample_bylevel = Colsample_bylevel +0.1
#Colsample_bynode = Colsample_bynode +0.1
if i == 1:
    irrelevant = 1
#    MaximumDepth = 1
#    TreeMethod = 'exact'

if i > 1:
    irrelevant = 2
 #   Grow_Policy = 'depthwise'
 #   MaximumDepth += 1
 #   TreeMethod = 'approx'
 #   Updater = 'prune'
```

# Multiple Parameters changed

```
def find_all(y_test_f,X_test_f,model_f,X_train_f, y_train_f):
    model_f.fit(X_train_f, y_train_f)
    preds = model_f.predict_proba(X_test_f)[:,1]
    fpr, tpr, thresholds  = metrics.roc_curve(y_test_f, preds)
    roc_auc = metrics.auc(fpr, tpr)
    y_pred_f = model_f.predict(X_test_f)
    cf = confusion_matrix(y_test_f, y_pred_f)
    accuracy_score_train =  model_f.score(X_train_f, y_train_f)
    accuracy_score_test =  model_f.score(X_test_f, y_test_f)
    return{'auc': roc_auc, 'cfm':cf ,'accuracy_score_train':accuracy_score_train , 'accura
```

# For each iteration printed

```python
## Below is Printing
print(color.BOLD +" ", i ,") XG_Results")
print ('\033[0m')
print(" ")
print( "learning_rate",  LearningRate)
print("process_type ", Process_type)
print ("Updater ", Updater)
print("max_depth", MaxDepth)
print( "grow_policy",  Grow_Policy)
print("tree_method",  TreeMethod)
print( "colsample_bytree", Colsample_bytree, "colsample_bylevel", Colsample_bylevel , "colsample_bynode", Co
print(" ")
print("XG_Results AUC Test  %.2f%%" % (XG_Results2['auc']* 100.0))
print(XG_Results2['cfm'])
print("XG_Results accuracy_score_train  %.2f%%" % (XG_Results2 ['accuracy_score_train'] * 100.0))
print("XG_Results accuracy_score_test  %.2f%%" % (XG_Results2 ['accuracy_score_test']* 100.0))
print(" ")
```

```
 4 ) XG_Results


learning_rate 0.4
process_type  default
max_depth 4
grow_policy depthwise
tree_method approx
colsample_bytree 0.6 colsample_bylevel 0.6 colsample_bynode 0.6

XG_Results AUC Test  81.88%
[[913 148]
 [157 191]]
XG_Results accuracy_score_train  81.66%
XG_Results accuracy_score_test  78.35%
```

# Ran some AB tests to see what changed

## NO CHANGE

grow_policy= 'lossguide’ -> depthwise
tree_method= 'auto’ - > exact -> approx
colsample_bytree= 0.3 -> 1
colsample_bynode= 0.3 - >0.6

## RESULTS CHANGE

colsample_bytree= 1 -> 0.5 AUC down 0.1%
learning_rate= 0.2 -> 0.4 AUC up 0.5%
colsample_bylevel= 0.3 - >0.6 AUC down 0.1%

Eg (colsample_bytree= 0.5, learning_rate= 0.1, max_depth= 2, grow_policy= 'depthwise', colsample_bylevel= 0.3, colsample_bynode= 0.6, tree_method= 'approx', process_type= 'default', verbosity = 1)

# Can see impact of all changes OR 1 at a time

```
 1 ) XG_Results


learning_rate 0.1
process_type  default
max_depth 1
grow_policy lossguide
tree_method auto
colsample_bytree 0.3 colsample_bylevel 0.3 colsample_bynode 0.3

XG_Results AUC Test  83.80%
[[1009   52]
 [ 232  116]]
XG_Results accuracy_score_train  76.89%
XG_Results accuracy_score_test  79.84%
```

```
 2 ) XG_Results


learning_rate 0.2
process_type  default
max_depth 2
grow_policy lossguide
tree_method exact
colsample_bytree 0.4 colsample_bylevel 0.4 colsample_bynode 0.4

XG_Results AUC Test  84.44%
[[941 120]
 [159 189]]
XG_Results accuracy_score_train  79.02%
XG_Results accuracy_score_test  80.20%
```

```
 5 ) XG_Results


learning_rate 0.5
process_type  default
max_depth 5
grow_policy depthwise
tree_method approx
colsample_bytree 0.7 colsample_bylevel 0.7 colsample_bynode 0.7

XG_Results AUC Test  79.18%
[[884 177]
 [153 195]]
XG_Results accuracy_score_train  86.39%
XG_Results accuracy_score_test  76.58%
```

```
 6 ) XG_Results


learning_rate 0.6
process_type  default
max_depth 6
grow_policy depthwise
tree_method approx
colsample_bytree 0.7999999999999999 colsample_bylevel 0.7999999999999999 colsampl

XG_Results AUC Test  77.19%
[[862 199]
 [152 196]]
XG_Results accuracy_score_train  90.61%
XG_Results accuracy_score_test  75.09%
```

# Changing 1 Variable

```
2 ) XG_Results

learning_rate 0.2
process_type  default
max_depth 1
grow_policy lossguide
tree_method auto
colsample_bytree 0.3 colsample_bylevel 0.3 colsample_bynode 0.3

XG_Results AUC Test  84.41%
[[956 105]
 [170 178]]
XG_Results accuracy_score_train  78.04%
XG_Results accuracy_score_test  80.48%
```

```
3 ) XG_Results

learning_rate 0.30000000000000004
process_type  default
max_depth 1
grow_policy lossguide
tree_method auto
colsample_bytree 0.3 colsample_bylevel 0.3 colsample_bynode 0.3

XG_Results AUC Test  84.62%
[[939 122]
 [162 186]]
XG_Results accuracy_score_train  78.24%
XG_Results accuracy_score_test  79.84%
```

```
4 ) XG_Results

learning_rate 0.4
process_type  default
max_depth 1
grow_policy lossguide
tree_method auto
colsample_bytree 0.3 colsample_bylevel 0.3 colsample_bynode 0.3

XG_Results AUC Test  84.68%
[[931 130]
 [158 190]]
XG_Results accuracy_score_train  78.31%
XG_Results accuracy_score_test  79.56%
```

Best

```
5 ) XG_Results

learning_rate 0.5
process_type  default
max_depth 1
grow_policy lossguide
tree_method auto
colsample_bytree 0.3 colsample_bylevel 0.3 colsample_bynode 0.3

XG_Results AUC Test  84.65%
[[930 131]
 [158 190]]
XG_Results accuracy_score_train  78.56%
XG_Results accuracy_score_test  79.49%
```

Starts getting worse

# Thanks

## Alex Dance

### Background

- Maths / statistics degree
- Background in big data, strategy, analytics
- Worked at Optus, Salmat, Reuters, Pathfinder Solutions

### Copy of This Presentation and code

https://github.com/alexdance2468/

Plus other data science projects completed

### Contact Details

www.linkedin.com/in/alex-dance/