Modeling Sales Win Rates based on historical CRM data

Sales forecasting has always been difficult and is often based on the gut feeling of sales leaders. I took a qualitative approach to predict whether sales opportunities will be won or lost by building a machine learning model trained on historical sales data.

This notebook trains a model that predicts if a sales opportunity will be won or lost. The optimized model has a recall of 0.84. Recall was selected as the metric since the real world dataset is imbalanced, and it's most important to locate true positives (winnable deals). Overall, it's a pretty good model and could be further improved by increasing the number of estimators but that would increase training time.

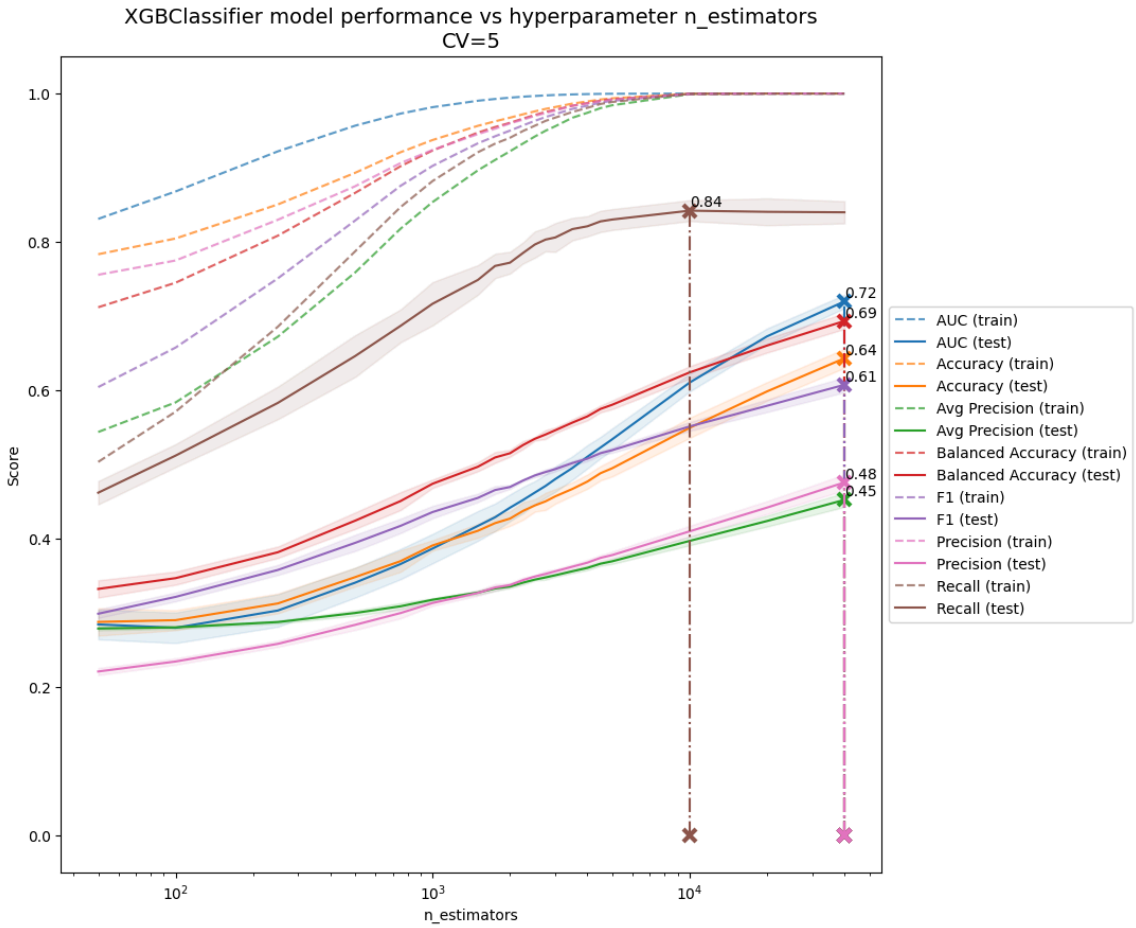
Optimized Metric	Value
Recall	0.84
Accuracy	0.79
AUC	0.77
Balanced Accuracy	0.70
F1	0.62
Log Loss	0.86
Avg Precision	0.46

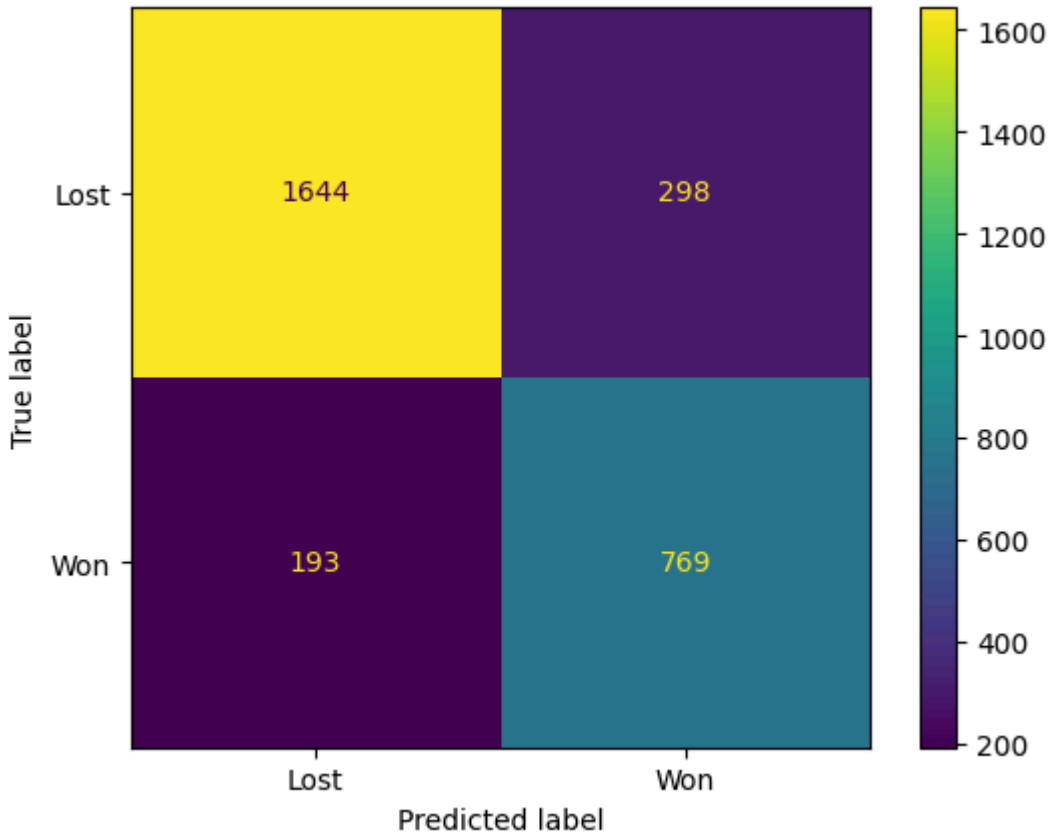
This notebook generates a variety of XGBoost Classifier models and selects hyperparameters to optimize the Recall metric.

What I would do differently next time

I would spend more time in exploratory data analysis and rebalancing the data before modeling. The first metric I used was accuracy, which was a mistake since about 80% of all the deals were lost, so it was a very imbalanced. I could have written one line of code, return False, that would have had an accuracy of 0.8 but that wouldn't have been useful. Normalizing numeric values increased recall from 0.7 to 0.8, so I should have done that earlier. This was also the first project I implemnted using data pipelines, which would have saved me a lot of time on other projects.

Performance visualized:


XGBClassifier model performance vs hyperparameter n_estimators
CV=5
Score
n_estimators
0.84
0.72
0.69
0.64
0.61
0.48
0.45
AUC (train)
AUC (test)
Accuracy (train)
Accuracy (test)
Avg Precision (train)
Avg Precision (test)
Balanced Accuracy (train)
Balanced Accuracy (test)
F1 (train)
F1 (test)
Precision (train)
Precision (test)
Recall (train)
Recall (test)


1644
298
193
769
Lost
Won
True label
Lost
Won
Predicted label

Simulated opportunity data was generated using Python, an existing CRM dataset from Kaggle, and Google Sheets. No proprietary data or company secrets were used. The same code would work for almost any company's CRM export.

Future work ideas:

- Try a basic neural network classifier using Keras
- Investigate the individual sales rep's influence on the opp's win rate. Separate out "how winnable" is the opp and compare to how likely is this specific sales rep to win it?
- Implement predict_proba to give a probability of an opp being won.
- Simulate additional features such as:
  - time spent in each sales stage
  - POC success criteria
  - Partner names
  - sales rep tenure at the time of a deal
  - Date/month that a deal was opened
  - Source of opp/lead
- additional hyperparameter tuning:
  - adjusting threshold levels for binary classification
  - maybe write an outer loop to run a wide set of combinations

Install dependencies

Fixes issues with Lambda Labs instances. LambdaLabs uses an older version of JupyterLab, v4.0.9. Current is 4.2.1. Must upgrade all of these packages on Lambda Labs, otherwise it will cause errors

In [ ]:
```
%pip install --upgrade --quiet pip xgboost bottleneck pandas scikit-learn sc
print('\n\n========== Finished installing/upgrading PIP packages==========')
```

Load dependencies

In [ ]:
```
# Disable certain Lint checks that don't apply to Jupyter Notebooks, or that
# I just don't care about.
# pylint: disable=pointless-statement
# pylint: disable=fixme
# pylint: disable=expression-not-assigned
# pylint: disable=missing-module-docstring
# pylint: disable=invalid-name
# pylint: disable=import-error
# pylint: disable=line-too-long
```

```
import os
import importlib
import time
# import multiprocessing

import pandas as pd
import matplotlib.pyplot as plt
# import numpy as np
# import seaborn as sns

from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, accura

# My custom python module(s)
import saleslib             # my custom Python module for this project
importlib.reload(saleslib)  # force reload because I'm frequently making upd

# Module configuration
# sns.set_theme(style="darkgrid")    # causes minor issues with Confusion Ma
pd.options.display.max_rows    = 100    # display more for debugging
pd.options.display.max_columns = 100    # display more for debugging
```

GPU support

TODO: GPU is currently running 10x slower than CPU on LambdaLabs, need to investigate further

In [ ]:
```
# %pip install --upgrade pip setuptools conda requests_mock clyent==1.2.1
# %pip install cupy-cuda12x
# !python3 -m cupyx.tools.install_library --cuda 11.x --library cutensor
# %pip freeze | grep cupy
# import cupy
```

Setting custom parameters for this notebook

In [ ]:
```
# path to training data that was prepared by a separate notebook.
filename = os.path.join(
   os.getcwd(), "data", "raw_CRM_opps_export-dummydata_prepped.csv"
)

# percentage of training data to use for test
TEST_SIZE = 0.3

# random seed
from saleslib import RANDOM_STATE
import random
# TODO: investigate seeding in any other PRNG engines used, ensure consisten
random.seed(RANDOM_STATE)

# Number of cross validation folds
```

```
NUM_CV_FOLDS = 5

# the name for the column that indicates the label/target
# the training CSV file should have headers
LABEL_COLUMN_NAME = "Won"

# Which metric from the list below is used for optimizing in GridSearchCV
METRIC_TO_OPTIMIZE = "Recall"

# list of metrics to compute for each model in the GridSearch
# https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-para
scoring = {"Recall":             make_scorer(recall_score),
           "AUC":                "roc_auc",                              #
           "Accuracy":           make_scorer(accuracy_score),            #

           "F1":                 make_scorer(f1_score),                  #
           "Balanced Accuracy":  make_scorer(balanced_accuracy_score),   #
           "Precision":          make_scorer(precision_score),           #
           "Avg Precision":      make_scorer(average_precision_score),    #
           #"neg_log_loss":       "neg_log_loss",
           }
```

Load, validate, split up the training data

Check that the data does not contain any missing values and all values are numeric datatypes.

In [ ]:
```
df = pd.read_csv(filename, header=0)
saleslib.verify_data_ready_for_training(df)
y = df[LABEL_COLUMN_NAME]
X = df.drop(columns=LABEL_COLUMN_NAME, axis=1)

# save memory since this variable takes up a lot of space and isn't used aga
del df

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=TEST_SIZE, random_state=RANDOM_STATE,
    #stratify=y      # force stratification across target column
)
```

Define the search grid and run it

https://xgboost.readthedocs.io/en/latest/python/python_api.html

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

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

In [ ]:
```
generic_model = XGBClassifier(
    objective='binary:logistic',
```

```
    random_state=RANDOM_STATE,
    # booster='',        # options: gbtree (default, tree functions), gbline
    # tree_method='',    # options: hist (default, also called auto), approx
    # n_jobs=1,
    # device='cuda'      # force GPU usage instead of CPU
    )

# basic but broad search for major changes
# param_grid = {
#     "max_depth":     [2, 3, 4],
#     "n_estimators":  [100, 1000, 3200],
#     "learning_rate": [0.1, 0.5, 0.85]
#     }

# short test runs, just the optimal model
# param_grid = {
#     "max_depth":     [2],
#     "n_estimators":  [3200],
#     "learning_rate": [0.85]
#     }

# short test runs, just the optimal model
param_grid = {
    "max_depth":     [2],
    "n_estimators":  [3200],
    "learning_rate": [0.85]
    }

# for pretty charts:
# param_grid = {
#     "max_depth":     [2], #[2, 4, 8],
#     "n_estimators":  [50, 100, 250, 500, 750, 1000, 1500, 1750, 2000, 2250
#                       2500, 2750, 3000, 3500, 4000, 4500, 5000, 10000, 200
#                       40000],
#     "learning_rate": [0.85], #[0.01, 0.05, 0.1, 0.25, 0.35, 0.45, 0.55, 0.
#                      # 0.75, 0.8, 0.9]
#     }

# create the grid search object
clf = GridSearchCV(
    generic_model,
    param_grid=param_grid,
    verbose=1,
    n_jobs=-1,                      # use all available CPU cores
    cv=NUM_CV_FOLDS,
    scoring=scoring,                # list of all metrics to compute
    refit=METRIC_TO_OPTIMIZE,       # list of metric to optimize
    return_train_score=True         # used in charts below
)

gridsearch_time_start = time.time() # start a timer for the grid search
# clf.fit(cupy.array(X), y)         # GPU version
clf.fit(X, y)                       # CPU version
gridsearch_time_stop = time.time()  # end the timer
gridsearch_time_total = gridsearch_time_stop - gridsearch_time_start
```

```
# Lambda Labs doesn't stream subprocesses to Jupyter Notebooks, so it is
# hard to see progress or know when it's done, even with verbose=4
#print('\n==================== FIT FINISHED ================================

#.cv_results_ does not offer training times on specific models or on the gri
# search as a whole.
print(f'* Total Grid Search time: {gridsearch_time_total:.0f} sec')
```

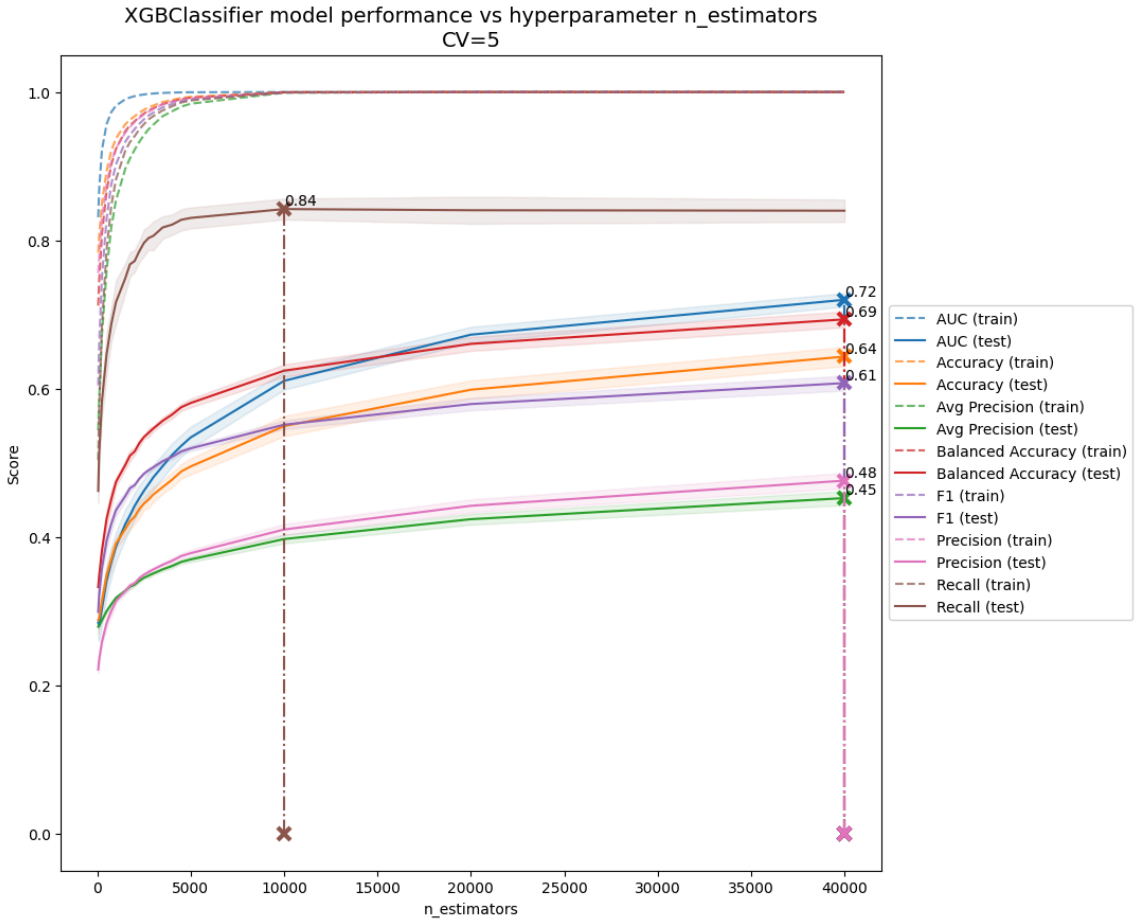
Display results of hyperparmater search

Summarize which parameters were searched and how the optimal model performed. Stores the optimal model for later.

In [ ]:
```
bst = saleslib.train_and_evaluate_optimal_model_from_gridsearch(clf,
                                          METRIC_TO_OPTIMIZE, X_train, y_train
```

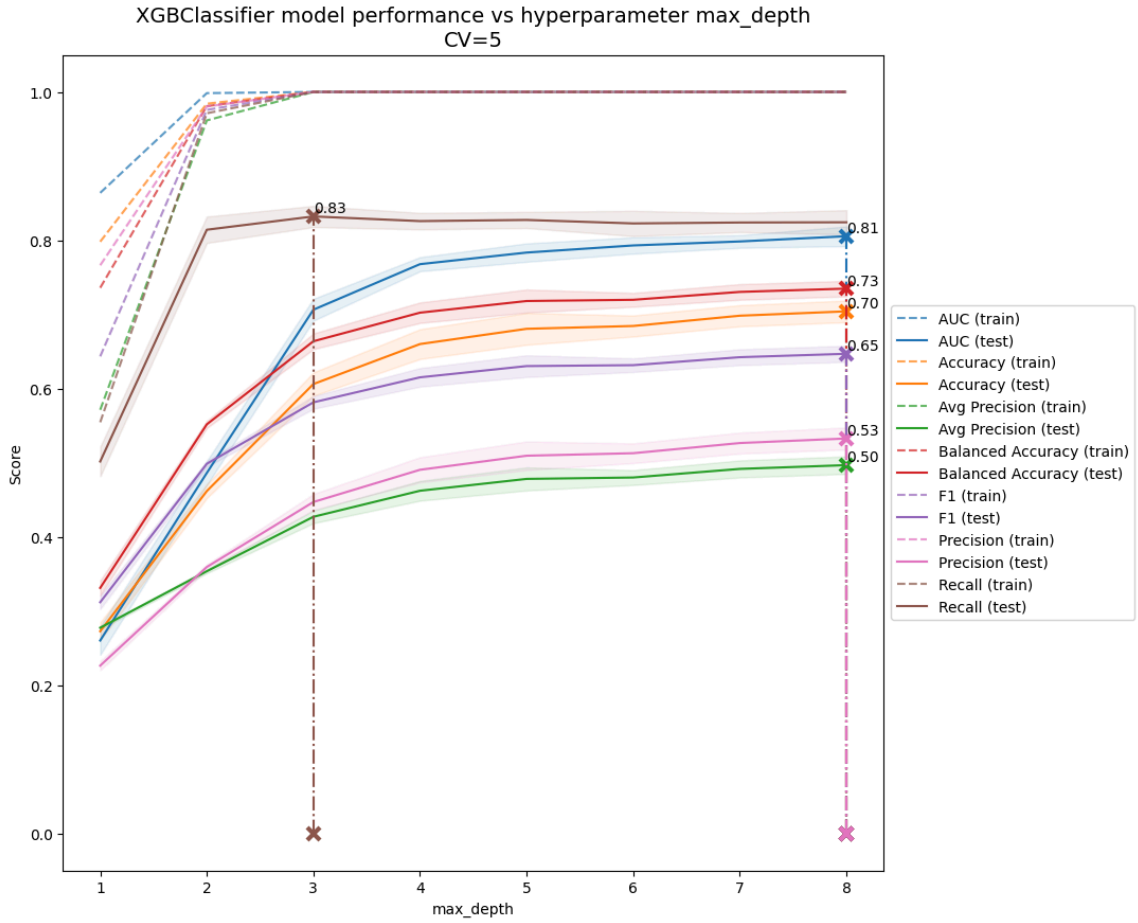
Visualizing model performance for each hyperparameter

Chart - Performance vs hyperparameter n_estimators



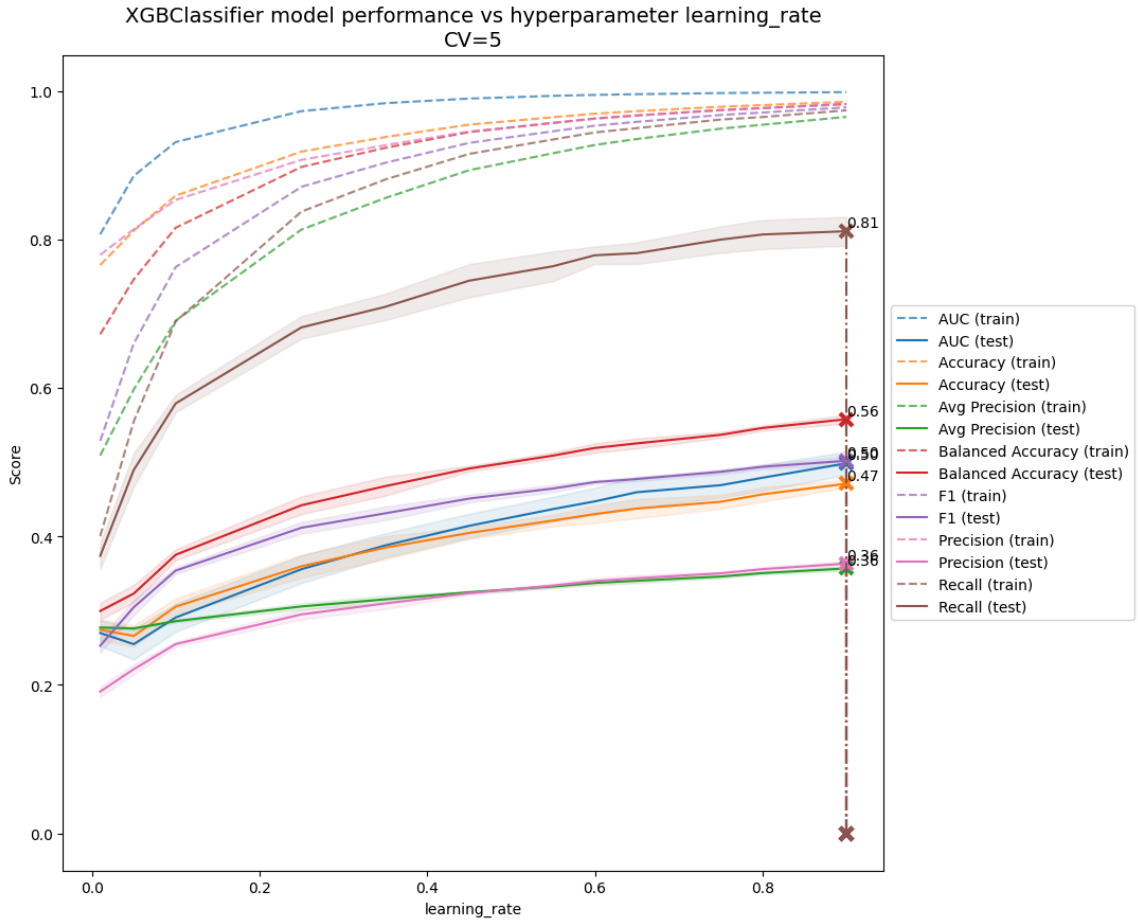
In [ ]:
```
saleslib.plot_grid_search_scores('n_estimators', clf, scoring)  #, scale='lo
```

Chart - Performance vs hyperparameter max_depth



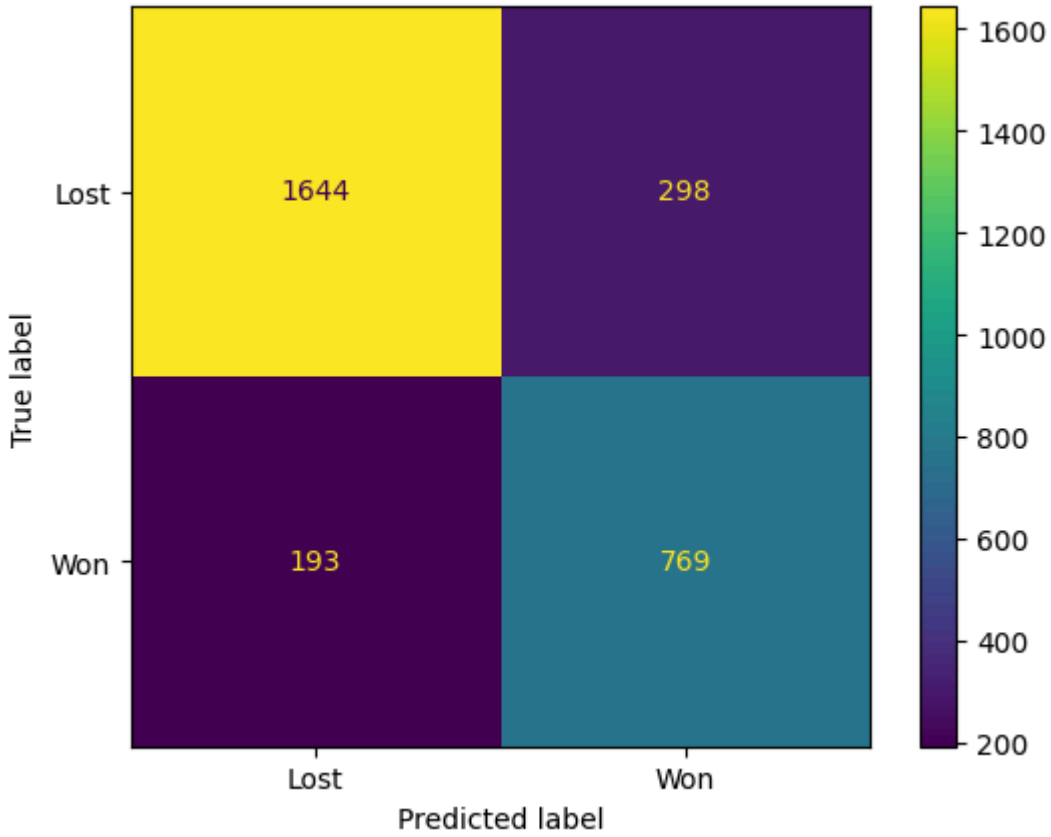
In [ ]:
```
saleslib.plot_grid_search_scores('max_depth', clf, scoring)
```

Chart - Performance vs hyperparameter learning_rate



In [ ]:
```
saleslib.plot_grid_search_scores('learning_rate', clf, scoring)
```

Confusion Matrix



In [ ]:

```
preds = bst.predict(X_test)
cm    = confusion_matrix(y_test, preds, labels=bst.classes_)

disp  = ConfusionMatrixDisplay(confusion_matrix=cm,
                               display_labels=['Lost', 'Won'])

disp.plot()

# another option for displaying with different color gradients
# ConfusionMatrixDisplay.from_estimator(clf, X_test, y_test, cmap='Greens')

plt.show()
```