Advanced Classification - Part 4

One should look for what is and not what he thinks should be. (Albert Einstein)

Warm up

- Before we continue, check out these examples of how machine learning is used: ***link***
- In the chat, answer the following questions:
  - How do you use machine learning?
  - How do you plan to use methods and tools we've learned about so far? Think about specific tasks and goals you would like to achieve.

Module completion checklist

Objective	Complete
Implement base GBM model and assess its performance	
Optimize GBM model using RandomizedCV method	
Use performance metrics to compare all ensemble methods	

Loading packages

- Let’s load the packages we will be using in this module

```
# Helper packages.
import os
import pickle
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import math
from pathlib import Path

# Scikit-learn packages for building models and model evaluation.
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn import metrics
```

Recap: boosting intuition

- In simple linear regression, you can clearly see the residuals, which are the multiple points around the linear model
- Let's think of these residuals, but apply the concept to decision trees
- When **gradient boosting** uses decision trees, it follows these three steps:
  - It **sees the errors** from a decision tree on the dataset
  - **Identifies the pattern of errors** and builds a new decision tree on them
  - It **repetitively leverages these patterns** in residuals to strengthen the overall model

Gradient boosted trees

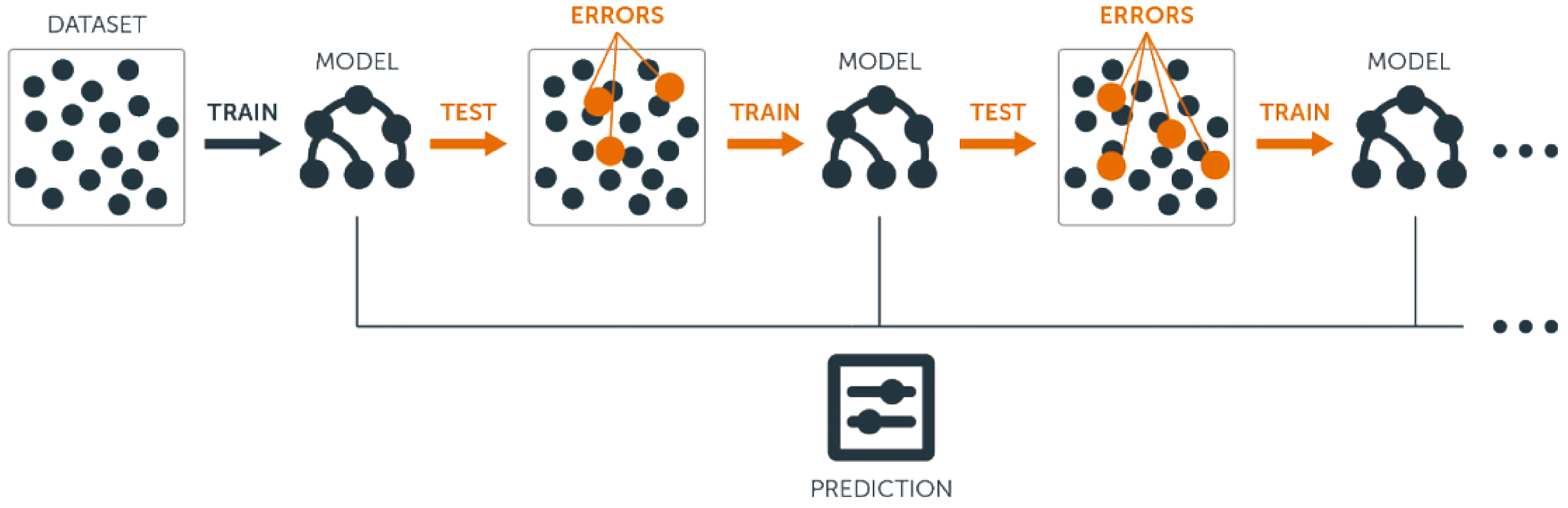


Image source

Recap: boosting process

- **Gradient boosting** can be used with classification or regression
- The three simple steps of **gradient boosting**
  - Fit a decision tree model to the data
  - Fit a decision tree model to the residuals
  - Create a new model
- The generalization of the multiple weak learners occurs by the optimization of a differentiable **loss function**
- The **loss function** will change based on the model's target variable:
  - **Regression**: ***gradient descent*** used to minimize MSE
  - **Binary classification**: ***logistic function***, yes, the same one we use for estimation of **log loss**!

Review data cleaning steps

- **Today, we will be loading the cleaned dataset we used earlier**
- To recap, the steps to get to this cleaned dataset were:
    - Remove household ID and individual ID
    - Remove variables with over 50% NAs
    - Transformed target variable to binary

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- We will use the `pathlib` library
- Let the `main_dir` be the variable corresponding to your `course` folder
- Let `data_dir` be the variable corresponding to your `data` folder

```
# Set 'main_dir' to location of the project folder
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

Load the cleaned dataset and model metrics

- Let's load the dataset: `costa_clean`
- Assign it to `costa_clean` variable

```
costa_clean = pickle.load(open(data_dir + "/costa_clean.sav","rb"))
metrics_gbm = pickle.load(open(data_dir + "/metrics_forest.sav","rb"))
print(costa_clean.head())
```

```
   rooms  tablet  males_under_12  ...  rural_zone  age  Target
0      3       0               0  ...           0   43    True
1      4       1               0  ...           0   67    True
2      8       0               0  ...           0   92    True
3      5       1               0  ...           0   17    True
4      5       1               0  ...           0   37    True

[5 rows x 81 columns]
```

Print info for our data

- Let’s view the column names

```
costa_clean.columns
```

```
Index(['rooms', 'tablet', 'males_under_12', 'males_over_12', 'males_tot',
       'females_under_12', 'females_over_12', 'females_tot', 'ppl_under_12',
       'ppl_over_12', 'ppl_total', 'years_of_schooling', 'wall_block_brick',
       'wall_socket', 'wall_prefab_cement', 'wall_wood', 'floor_mos_cer_terr',
       'floor_cement', 'floor_wood', 'ceiling', 'electric_public',
       'electric_coop', 'toilet_sewer', 'toilet_septic', 'cookenergy_elec',
       'cookenergy_gas', 'trash_truck', 'trash_burn', 'wall_bad', 'wall_reg',
       'wall_good', 'roof_bad', 'roof_reg', 'roof_good', 'floor_bad',
       'floor_reg', 'floor_good', 'disabled_ppl', 'male', 'female', 'under10',
       'free', 'married', 'separated', 'single', 'hh_head', 'hh_spouse',
       'hh_child', 'num_child', 'num_adults', 'num_65plus', 'num_hh_total',
       'dependency_rate', 'male_hh_head_educ', 'female_hh_head_educ',
       'meaneduc', 'educ_none', 'educ_primary_inc', 'educ_primary',
       'educ_secondary_inc', 'educ_secondary', 'educ_undergrad', 'bedrooms',
       'ppl_per_room', 'house_owned_full', 'house_owned_paying',
       'house_rented', 'house_other', 'computer', 'television',
       'num_mobilephones', 'region_central', 'region_Chorotega',
       'region_pacifico', 'region_brunca', 'region_antlantica',
       'region_huetar', 'urban_zone', 'rural_zone', 'age', 'Target'],
```

Split into training and test sets

```
# Select the predictors and target.
X = costa_clean.drop(['Target'], axis = 1)
y = np.array(costa_clean['Target'])

# Set the seed to 1.
np.random.seed(1)

# Split into training and test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)
```

Vanilla GBM model

```
# Initialize GBM model.
gbm = GradientBoostingClassifier()

# Fit the model to train data.
gbm.fit(X_train, y_train)
```

```
GradientBoostingClassifier()
```

Convenience function for performance metrics

```python
def get_performance_scores(y_test, y_predict, y_predict_prob, eps=1e-15, beta=0.5):
    from sklearn import metrics
    # Scores keys.
    metric_keys = ["accuracy", "precision", "recall", "f1", "fbeta", "log_loss", "AUC"]
    # Score values.
    metric_values = [None]*len(metric_keys)
    metric_values[0] = metrics.accuracy_score(y_test, y_predict)
    metric_values[1] = metrics.precision_score(y_test, y_predict)
    metric_values[2] = metrics.recall_score(y_test, y_predict)
    metric_values[3] = metrics.f1_score(y_test, y_predict)
    metric_values[4] = metrics.fbeta_score(y_test, y_predict, beta=beta)
    metric_values[5] = metrics.log_loss(y_test, y_predict_prob[:, 1], eps=eps)
    metric_values[6] = metrics.roc_auc_score(y_test, y_predict_prob[:, 1])
    perf_metrics = dict(zip(metric_keys, metric_values))
    return(perf_metrics)
```

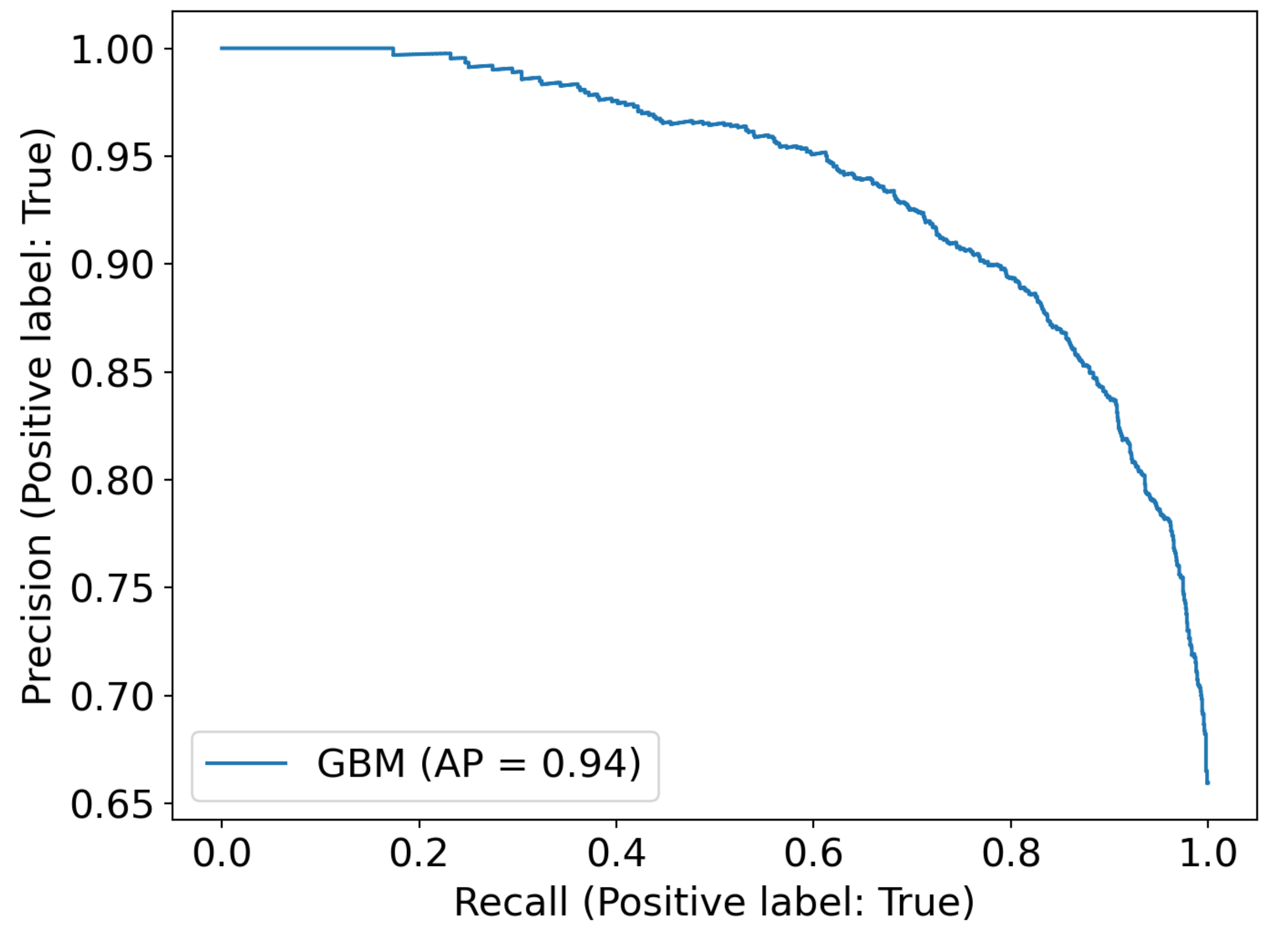
Predict and evaluate vanilla GBM model

```
# Predict on test data for GBM model.
gbm_y_predict = gbm.predict(X_test)

# Get prediction probabilities for the GBM model.
gbm_y_predict_proba = gbm.predict_proba(X_test)

# Get the GBM performance scores.
gbm_scores = get_performance_scores(y_test, gbm_y_predict, gbm_y_predict_proba)
```

Precision vs recall curve: the tradeoff

- Plot precision vs recall curve for GBM classifier

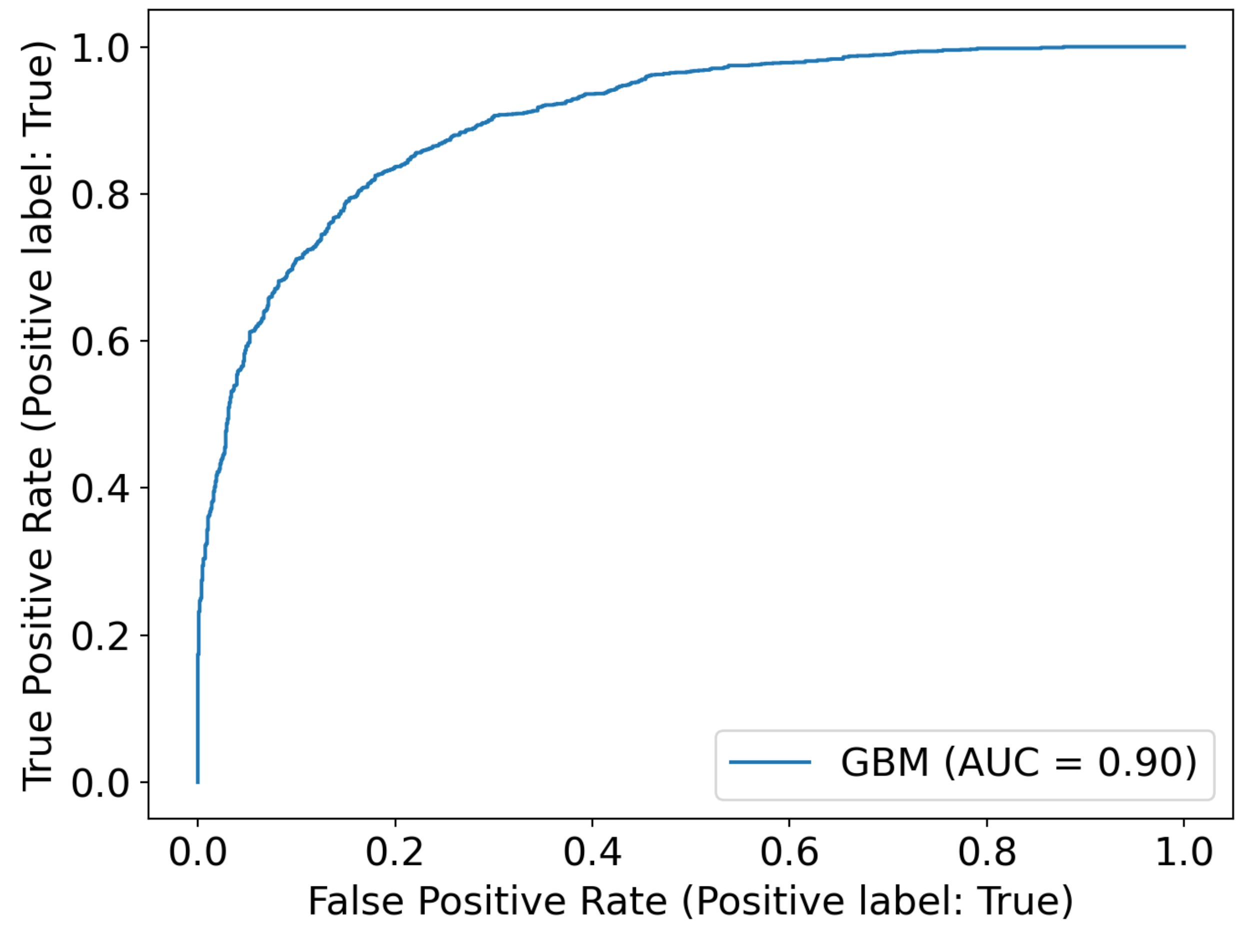
```
ax = plt.gca()
gbm_prec_recall = metrics.plot_precision_recall_curve(gbm,
                                    X_test,
                                    y_test,
                                    ax = ax,
                                    name = "GBM")

plt.show()
```



ROC curve: the tradeoff

```
ax = plt.gca()
gbm_roc = metrics.plot_roc_curve(gbm,
                                 X_test,
                                 y_test,
                                 ax = ax,
                                 name = "GBM")


plt.show()
```

- Looks like the ROC curve shows similar results
- Let's see if parameter tuning of the GBM can make it better



Append to other model performance scores

```
metrics_gbm.update({"GBM": gbm_scores})
print(metrics_gbm)
```

```
{'RF': {'accuracy': 0.9483960948396095, 'precision': 0.9447424892703863, 'recall':
0.9750830564784053, 'f1': 0.9596730245231607, 'fbeta': 0.9506586050529044, 'log_loss':
0.21947942349408847, 'AUC': 0.986855647527701}, 'Optimized RF': {'accuracy':
0.9483960948396095, 'precision': 0.9400212314225053, 'recall': 0.9806201550387597, 'f1':
0.9598915989159891, 'fbeta': 0.9478698351530723, 'log_loss': 0.21667604220201855, 'AUC':
0.9876901226920936}, 'GBM': {'accuracy': 0.8291492329149233, 'precision':
0.8374358974358974, 'recall': 0.9042081949058693, 'f1': 0.869542066027689, 'fbeta':
0.8499895898396836, 'log_loss': 0.39102395649143923, 'AUC': 0.902052011186816}}
```

Module completion checklist

Objective	Complete
Implement base GBM model and assess its performance	✓
Optimize GBM model using RandomizedCV method	
Use performance metrics to compare all ensemble methods	

Ways to optimize gradient boosting

- As with our RF model, we can use various parameter tuning techniques
- In this instance we will repeat the process of randomized grid search
- Since GBM parameters are almost identical to those of RF models, we will keep all of them the same except for one:
    - We will not use `ccp_alpha` - it will be set to default `0.0` value
    - We will instead tune `learning_rate` - the rate with which the error is corrected

Randomized CV for GBM optimization: parameters

```python
gbm = GradientBoostingClassifier()
# Number of trees in random forest.
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 20)]
# Number of features to consider at every split.
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree.
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node.
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node.
min_samples_leaf = [1, 2, 4]
# Define learning rate parameters.
learning_rate = [0.001, 0.01, 0.1, 0.2, 0.3]
```

Randomized CV for GBM optimization: create grid

```
# Create the random grid.
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'learning_rate': learning_rate}
```

Randomized CV for GBM optimization: fit

- Initialize randomized search for GBM
- Fit the model to the train data

```
# Initialize the randomized search model just as you would any other scikit model.
gbm_random = RandomizedSearchCV(estimator = gbm,
                                param_distributions = random_grid,
                                n_iter = 100,
                                cv = 3,
                                verbose = 0,
                                random_state = 1,
                                n_jobs = -1)

# Fit the random search model.
gbm_random.fit(X_train, y_train)
```

This function may run 30+ minutes! If you would like to skip this step, simply load the pre-saved model as shown below.

```
gbm_random = pickle.load(open(data_dir + "/gbm_random.sav","rb"))
```

```
gbm_random.best_params_
```

```
{'n_estimators': 578, 'min_samples_split': 5, 'min_samples_leaf': 4, 'max_features': 'auto',
'max_depth': 30, 'learning_rate': 0.1}
```

Implement optimized GBM model

- Now, we will run the optimized model on our `X_train`
- We will again pass the parameters to GMB classifier directly from the result of our randomized search by using the `**parameters` notation

```
# Pass parameters from randomized search to GBM classifier.
optimized_gbm = GradientBoostingClassifier(**gbm_random.best_params_)

# Fit model to train data.
optimized_gbm.fit(X_train, y_train)
```

```
GradientBoostingClassifier(max_depth=30, max_features='auto',
                           min_samples_leaf=4, min_samples_split=5,
                           n_estimators=578)
```

Knowledge check 1

Exercise 1

Module completion checklist

Objective	Complete
Implement base GBM model and assess its performance	✔
Optimize GBM model using RandomizedCV method	✔
Use performance metrics to compare all ensemble methods	

Predict and evaluate optimized GBM model

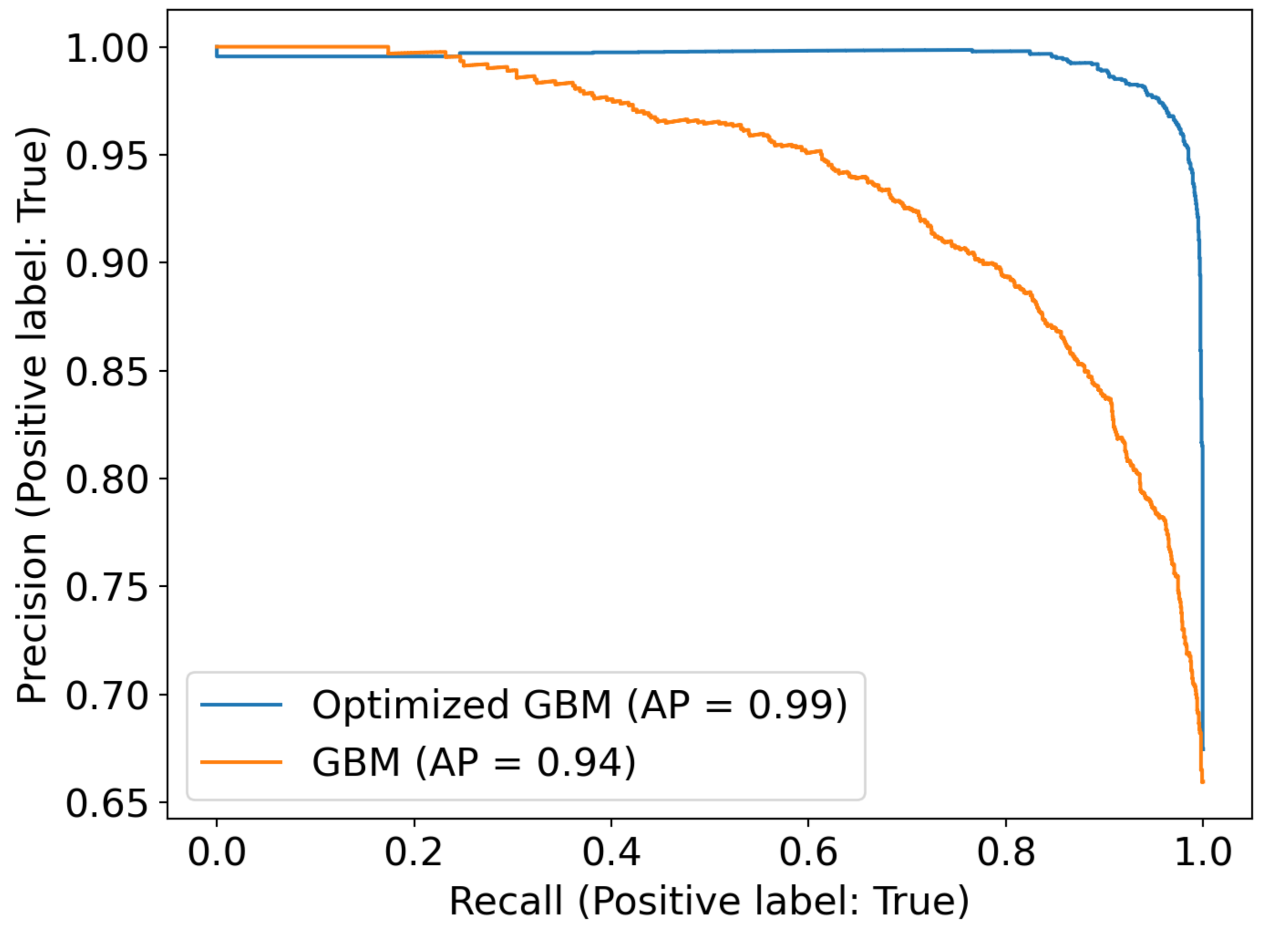
- Predict on test using the optimized GBM model
- Compute performance metrics

```
# Get class predictions.
optimized_gbm_y_predict = optimized_gbm.predict(X_test)

# Get prediction probabilities.
optimized_gbm_y_predict_proba = optimized_gbm.predict_proba(X_test)

# Compute performance metrics.
optimized_gbm_scores = get_performance_scores(y_test,
                                              optimized_gbm_y_predict,
                                              optimized_gbm_y_predict_proba)
```

Precision vs recall curve: the tradeoff

- Plot precision vs recall fore optimized GBM model
- Add all previous model curves to compare

```
ax = plt.gca()
opt_gbm_prec_recall =
metrics.plot_precision_recall_curve(optimized_gbm,
                                    X_test,
                                    y_test,
                                    ax = ax,
                                    name = "Optimized GBM")

gbm_prec_recall.plot(ax = ax, name = "GBM")
plt.show()
```

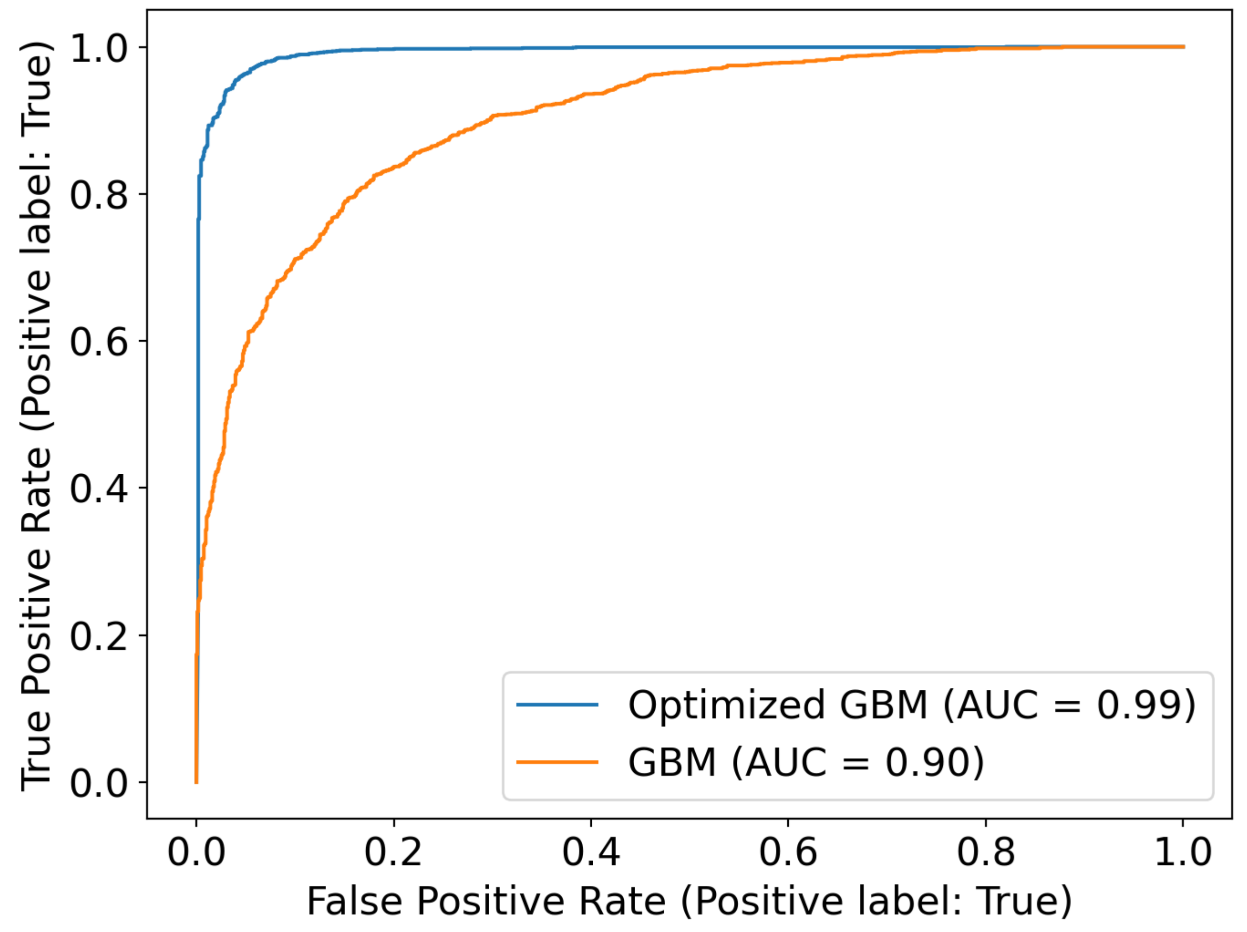


- We can clearly see how much better the optimized GBM performed compared to the vanilla GBM

ROC curve: the tradeoff

- Plot precision vs recall fore optimized GBM model
- Add all previous model curves to compare

```
ax = plt.gca()
opt_gbm_roc = metrics.plot_roc_curve(optimized_gbm,
                                     X_test,
                                     y_test,
                                     ax = ax,
                                     name = "Optimized
GBM")


gbm_roc.plot(ax = ax, name = "GBM")
plt.show()
```



- The same holds for the ROC curve and the AUC!

Append to other model performance scores

- Let's append optimized GBM scores to the dictionary and then take a look at them all next to each other

```
metrics_gbm.update({"Optimized GBM": optimized_gbm_scores})
print(metrics_gbm)
```

```
{'RF': {'accuracy': 0.9483960948396095, 'precision': 0.9447424892703863, 'recall':
0.9750830564784053, 'f1': 0.9596730245231607, 'fbeta': 0.9506586050529044, 'log_loss':
0.21947942349408847, 'AUC': 0.986855647527701}, 'Optimized RF': {'accuracy':
0.9483960948396095, 'precision': 0.9400212314225053, 'recall': 0.9806201550387597, 'f1':
0.9598915989159891, 'fbeta': 0.9478698351530723, 'log_loss': 0.21667604220201855, 'AUC':
0.9876901226920936}, 'GBM': {'accuracy': 0.8291492329149233, 'precision':
0.8374358974358974, 'recall': 0.9042081949058693, 'f1': 0.869542066027689, 'fbeta':
0.8499895898396836, 'log_loss': 0.39102395649143923, 'AUC': 0.902052011186816}, 'Optimized
GBM': {'accuracy': 0.9592050209205021, 'precision': 0.9678670360110804, 'recall':
0.9673311184939092, 'f1': 0.9675990030462477, 'fbeta': 0.9677598050077555, 'log_loss':
0.5621060823499561, 'AUC': 0.9918591095177615}}
```

Convert metrics dictionary to dataframe

- Let's convert our dictionary to a dataframe

```
# Convert all metrics for each model to a dataframe.
metrics_gbm_df = pd.DataFrame(metrics_gbm)
metrics_gbm_df["metric"] = metrics_gbm_df.index
metrics_gbm_df = metrics_gbm_df.reset_index(drop = True)
print(metrics_gbm_df.head())
```

```
         RF  Optimized RF       GBM  Optimized GBM     metric
0  0.948396      0.948396  0.829149       0.959205   accuracy
1  0.944742      0.940021  0.837436       0.967867  precision
2  0.975083      0.980620  0.904208       0.967331     recall
3  0.959673      0.959892  0.869542       0.967599         f1
4  0.950659      0.947870  0.849990       0.967760      fbeta
```

Convert wide to long format

- To help with plotting, let's switch our wide dataframe to long format using `melt` function

```
metrics_gbm_long = pd.melt(metrics_gbm_df,
                           id_vars = "metric",
                           var_name = "model",
                           value_vars = list(metrics_gbm.keys()))
print(metrics_gbm_long.head())
```

```
      metric model     value
0   accuracy    RF  0.948396
1  precision    RF  0.944742
2     recall    RF  0.975083
3         f1    RF  0.959673
4      fbeta    RF  0.950659
```

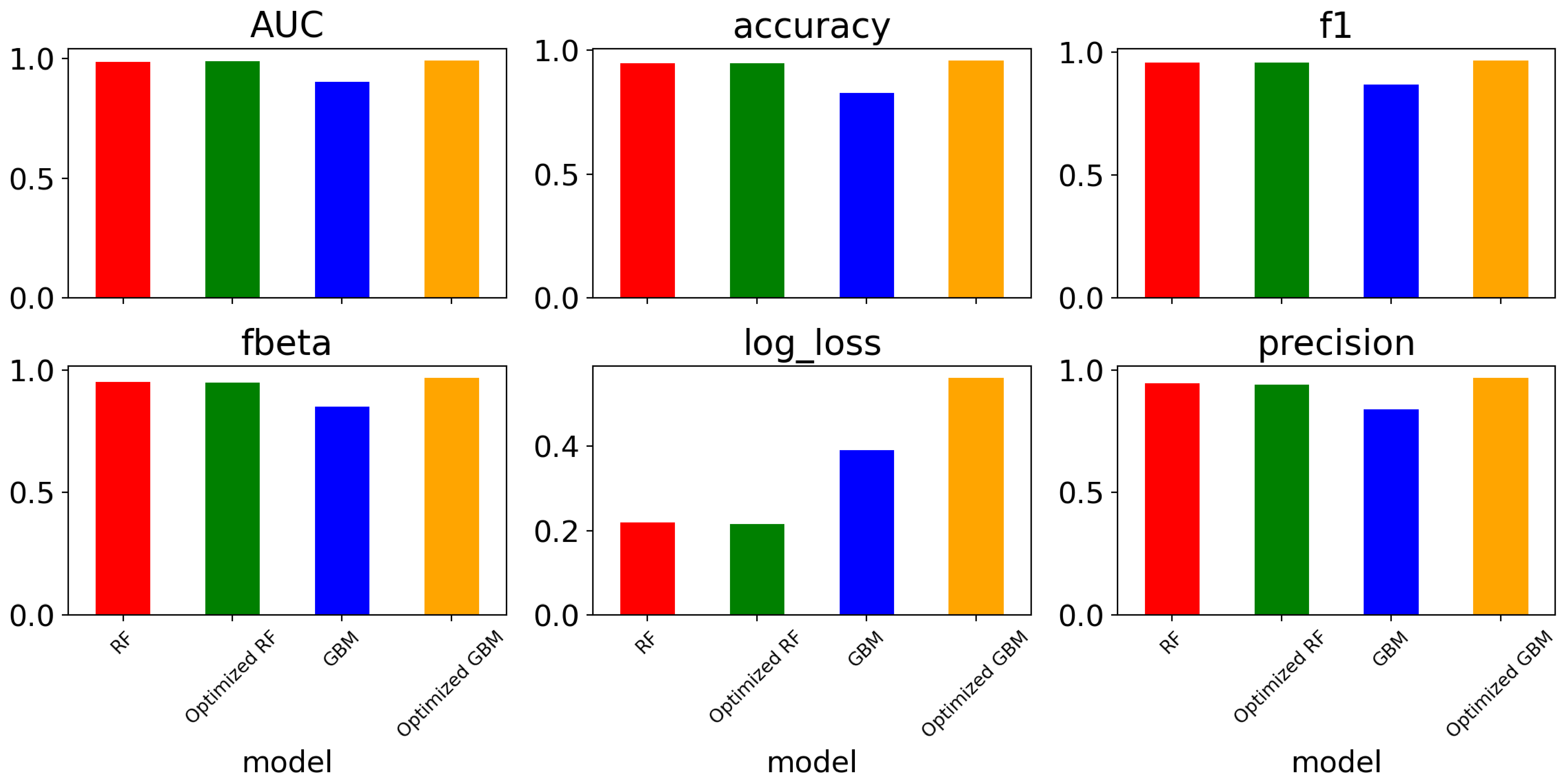
- Take a look at the documentation ***here*** for more details on this function

Plot all models by metric

- Since we converted our metrics dataframe to `long` format, it's now easy to plot each model grouped by metric in a grid
- Each plot within a grid will be a bar chart with as many bars as there are models

```
# Create a 2X3 grid.
fig, axes = plt.subplots(2, 3, figsize = (12, 6))
# For each group in a grouped by metric dataframe, assign a metric to an axis object.
for (metric, group), ax in zip(metrics_gbm_long.groupby("metric"), axes.flatten()):
    # Plot each metric as a bar plot.
    group.plot(x = 'model',                                      #<- model on x-axis
               y = 'value',                                      #<- metric value on y-axis
               kind = 'bar',                                     #<- bar plot
               color = ["red", "green", "blue", "orange"],       #<- color for each model
               ax = ax,                                          #<- axis object
               title = metric,                                   #<- plot title
               legend = None,                                    #<- remove auto-legend
               sharex = True)                                    #<- use the same x-axis
    ax.xaxis.set_tick_params(rotation = 45, labelsize=10)                      #<- rotate labels for
prettiness
plt.tight_layout(0.5)                                            #<- make sure no space is unused
plt.show()
```

Plot all models by metric



Wrap comparison plot into a function

```
def compare_metrics(metrics_dict, color_list = None):
    metrics_df = pd.DataFrame(metrics_dict)
    metrics_df["metric"] = metrics_df.index
    metrics_df = metrics_df.reset_index(drop = True)
    metrics_long = pd.melt(metrics_df, id_vars = "metric", var_name = "model",
                           value_vars = list(metrics_dict.keys()))
    if color_list is None:
        cmap = plt.rcParams['axes.prop_cycle'].by_key()['color']
        colors = cmap[:len(metrics_dict.keys())]
    else:
        colors = color_list
    fig, axes = plt.subplots(2, 3, figsize = (12, 6))
    for (metric, group), ax in zip(metrics_long.groupby("metric"), axes.flatten()):
        group.plot(x = 'model', y = 'value', kind = 'bar', color = colors, ax = ax,
                   title = metric,legend = None,sharex = True)
        ax.xaxis.set_tick_params(rotation = 45, labelsize=10)
    plt.tight_layout(0.5)
    return((fig, axes))
```

Test function

```
fig, axes = compare_metrics(metrics_gbm)
```

```
plt.show()
```

Discuss model champion

- All metrics in this instance were championed by the optimized GBM model
- All but one were close to each other
- The log loss was noticeably lower for the optimized GBM
- We have our ensemble method winner - **optimized GBM** model!

Exercise 2

Module completion checklist

Objective	Complete
Implement base GBM model and assess its performance	✔
Optimize GBM model using RandomizedCV method	✔
Use performance metrics to compare all ensemble methods	✔

Next Steps

- So far, we have studied ensemble methods like Random Forest and Gradient Boosting Models for classification
- Every dataset is different and no one classification algorithm is a best fit for all types of datasets
- What if you have high dimensional data at work and you have to classify a target variable?
- What if your dataset contains more categorical variables?
- In the next module, we will learn about **Support vector machines** which were developed to handle such issues in our dataset

Congratulations on completing this module!