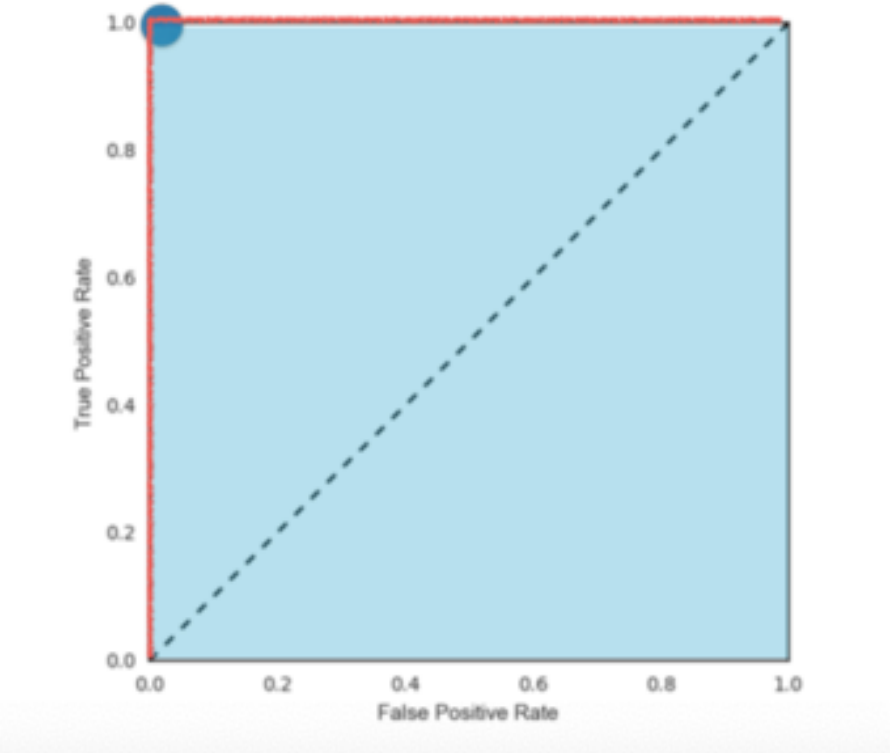
Extreme Gradient Boosting with XGBoost
by Sergey Fogelson on datacamp

Refresher on supervised learning
*relies on labeled data
*meaning we have some understanding of past behavior
majority of supervised learning problems are either classification or regression problems

We'll start with classification problems
can either predict binary or multi-class outcomes

For binary classification models the AUC (area under the receiver operating characteristic curve) is the most common tool
also possibly the most versatile
used to judge the quality of a binary classification model
it is simply the probability that a randomly chosen positive data point will have a higher rank than a randomly chosen negative data point for your learning problem
what this means?
a higher AUC means a more sensitive, better performing model



For multi-classification problems it is common to use the accuracy score
higher is better
here we look at the confusion matrix to evaluate the quality of a model

Common algorithms for classification problems include logistic regression and decision trees
all supervised learning problems require that the data is structured as a table of feature vectors
where the features (also called predictors or attributes) are either numeric or categorical
numerical features are scaled to aid in feature interpretation
also scaling features ensures that the model can be trained properly (essential for SVM models)
categorical features almost always are encoded
most common route is through one-hot encoding

Examles of some other kind of supervised learning problems
ranking > predicting an ordering on a set of choices (ie like Google search suggestions)
recommendation > recommending based on consumption (like Netflix)

XGBoost
core algorithm is parallelizable
ie it can harness all of the processing power of modern multi-core computers
can use this trait across GPUs or networks
however its main popularity stems from its ability to consistently outperform other models

example
```
import xgboost as xgb
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
class_data = pd.read_csv('classification_data.csv')

#split entire dataset into a matrix of samples by features called X
#and a vector of target values called y
X, y = class_data.iloc[:,:-1], class_data.iloc[:,-1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=123)
xg_cl = xgb.XGBClassifier(objective='binary:logistic', n_estimators=10, seed=123)
sg_cl.fit(X_train, y_train)
```

preds = xg_cl.predict(X_test)
accuracy - float(np.sum(pres==y_test))/y_test.shape[0]
print('accuracy: %f' % (accuracy))

XGBoost is often used with decision trees
key things with decision trees
base learners > meaning individual learning algorithm in an ensemble algorithm
composed of a series of binary questions
XGBoost in itself is an ensemble learning method
*it uses the outputs of many models for a final prediction
decision trees are constructed iteratively (that is one binary decision at a time) until some stopping criterion is met
the tree is built on split points
these split points put each target category into buckets that are increaslingly dominated by just one category
this continues until all or nearly all values within a given split are exclusively of one category or another
**individual decision trees are usually high variance, low bias learning models
meaning they are very good at learning relationships on training set but in the process tend to overfit
which then tend to generalize poorly to new data
XGBoost uses a special kind of decision tree called CART (classification and regression tree)
these trees instead of containing decision values contain real-valued scores

example
```
# Import the necessary modules
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

# Create the training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=123)

# Instantiate the classifier: dt_clf_4
dt_clf_4 = DecisionTreeClassifier(max_depth=4)

# Fit the classifier to the training set
dt_clf_4.fit(X_train, y_train)

# Predict the labels of the test set: y_pred_4
y_pred_4 = dt_clf_4.predict(X_test)
```

# Compute the accuracy of the predictions: accuracy
accuracy = float(np.sum(y_pred_4==y_test))/y_test.shape[0]
print("accuracy:", accuracy)

So what is boosting?
not really an ML algorithm
more a concept that can be applied to a set of ML models
dubbed an ‘ensemble meta-algorithm’
primarily used to reduce any given single learner’s variance
and to convert many weak learners into a strong learner

What is a weak learner?
an algorithm that is slightly better than chance (predictions slightly greater than 50%)

How boosting is accomplished?
by iteratively learning a set of weak models on subsets of data and weighing each of their predictions based on performance
then combine all of the weak learner’s predictions multiplied by their weights to obtain a single final weighted prediction that is better than any of the individual predictions

*XGBoost’s learning API is different from scikit’s
XBG in addition uses cross-validation
cross-validation is a robust method for estimating the expected performance of an ML model on unseen data
does this by generating many non-overlapping train/test splits on your training data
then reports the average test set performance across all data splits

example - churn rate at 5 months
import xgboost as xgb
import pandas as pd
churn_data - pd.read_csv(‘classification)data.csv”)
#with XGBoost API we need to convert our dataset into an optimized data structure called a DMatrix
#with scikit API this is taken care of
#can look at it as data = X and label = y
churn_dmatrix = xgb. DMatrix(data=churn_data.iloc[:,:-1],
label=churn_data.month_5_still_here)
#required to create a dictionary to pass into our cv (cross validation) method
params = {‘objective’:’binary”logistic’, ‘max_depth’:4}
#n_folds is how many cv folds

```
#num_boost_round is how many trees we want to build
#metrics is what we want to compute
#as_pandas gives us the option to store our output as a pandas dataframe
cv_results = xgb.cv(dtrain=churn_dmatrix, params=params, nfold=4,
num_boost_round=10, metrics='error', as_pandas=True)
print('Accuracy: %f' %((1-cv_results['test-error-mean']).iloc[-1]))

# Create arrays for the features and the target: X, y
X, y = churn_data.iloc[:,:-1], churn_data.iloc[:,-1]

# Create the DMatrix from X and y: churn_dmatrix
churn_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params
params = {"objective":"reg:logistic", "max_depth":3}

# Perform cross-validation: cv_results
cv_results = xgb.cv(dtrain=churn_dmatrix, params=params,
                    nfold=3, num_boost_round=5,
                    metrics="error", as_pandas=True, seed=123)

# Print cv_results
print(cv_results)

# Print the accuracy
print(((1-cv_results["test-error-mean"]).iloc[-1]))

# Perform cross_validation: cv_results
cv_results = xgb.cv(dtrain=churn_dmatrix, params=params,
                    nfold=3, num_boost_round=5,
                    metrics="auc", as_pandas=True, seed=123)

# Print cv_results
print(cv_results)

# Print the AUC
print((cv_results["test-auc-mean"]).iloc[-1])
```

When should we use XGBoost?
you have a large number of training samples
accepted threshold is greater than 1000 training samples and less than 100 features
*key to remember features should be less than the number of examples you have

XGB does well when you have a mix of categorical and numeric features
or if you just have numeric features

When not to use?
small training sets
or number of training examples is significantly smaller than the number of features being used for training
falls short to deep learning approaches in:
-NLP
-computer vision
-image recognition

Moving onto XGB for regression problems
regression problems involve predicting continuous, or real, values

Common regression metrics to evaluate quality of a regression model
-root mean squared error (RMSE)
refresher > take difference between actual and predicted, squaring those differences, computing their mean, then taking that value's square root
we take square so the negatives and positive do not cancel out
also punishes larger differences
-mean absolute error (MAE)
refresher > sums the absolute differences between predicted and actual values across all of the samples

Common regression algorithms
-linear regression
-decision trees
**key note decision trees can be used for both regression and classification tasks this is one of the reasons they are prime candidates to be building blocks for XGBoost models

Objective (also called loss) function
quantifies how far off our prediction is from the actual result for a given data point
maps the difference between the prediction and the target to a real number
*when we construct any ML model, we do so in the hopes that it minimizes the loss function across all of the data points we pass in
**the ultimate goal is the smallest possible loss

Common loss functions for XGBoost
reg:linear > for regression problems
reg:logistic > for binary classification models (most common)
*use for classification problems when you want just decision, not probability

when you want the category of the target
binary:logistic > when you want the actual predicted probability of the positive class

XGBoost is a meta-model that is composed of many individual models that combine to give a final prediction
these individual models are called base learners
want base learner that when combined create final prediction that is non-linerar
this means base learners that are slightly better than random guessing on certain subsets of training examples,
and uniformly bad at the remainder
this is so when all the predictions are combined, the uniformly bad predictions cancel out,
and those slightly better than chance combine into a single very good prediction
can have tree base learners or linear base learners

example
#convert data into X matrix and y vector
X, y = boston_data.iloc[:, :-1], boston_data.iloc[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
xg_reg = xgb.XGBRegressor(objective='reg:linear', n_estimators=10, seed=123)
xg_reg.fit(X_train, y_train)
preds = xg_reg.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, preds))
print('RMSE: %f; % (rmse))

for linear base learners we have to use the learning API in XGBoost
X, y = boston_data.iloc[:, :-1], boston_data.iloc[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
#the difference, convert training and test sets into DMatrix objects
DM_train = xgb.DMatrix(data=X_train, label=y_train)
DM_test = xgb.DMatrix(data=X_test, label=y_test)
#create a parameter dictionary specifying base learner we want as gblinear and reg:linear as objective (loss) function
params = {'booster':'gblinear', 'objective':'reg:linear'}
xg_reg = xgb.train(params=params, dtrain=DM_train, num_boost_round=10)
preds = xg_reg.predict(DM_test)
rmse = np.sqrt(mean_squared_error(y_test, preds))
print('RMSE: %f' % (rmse))

XGBoost by default uses trees as base learners

argument ‘booster’ allows you to change the base learner

Example

```
# Convert the training and testing sets into DMatrixes: DM_train, DM_test
DM_train = xgb.DMatrix(data=X_train, label=y_train)
DM_test =  xgb.DMatrix(data=X_test, label=y_test)

# Create the parameter dictionary: params
params = {"booster":"gblinear", "objective":"reg:linear"}

# Train the model: xg_reg
xg_reg = xgb.train(params=params, dtrain=DM_train, num_boost_round=5)

# Predict the labels of the test set: preds
preds = xg_reg.predict(DM_test)

# Compute and print the RMSE
rmse = np.sqrt(mean_squared_error(y_test,preds))
print("RMSE: %f" % (rmse))

# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params
params = {"objective":"reg:linear", "max_depth":4}

# Perform cross-validation: cv_results
cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=4,
num_boost_round=5, metrics='mae', as_pandas=True, seed=123)

# Print cv_results
print(cv_results)

# Extract and print final boosting round metric
print((cv_results["test-mae-mean"]).tail(1))
```

Regularization in XGBoost
Loss functions in XGBoost don’t just take into account how close a model’s prediction s are to the actual values
also take into accoun how complex the model is
the idea of penalizing models as they become more complex is called regularization
used to find models that are both simple and accurate

can tweak XGBoost model complexity by altering the loss function
gamma - for tree base learners, controls whether a given node on a base learner will split based on the expected reduction in the loss that would occur after performing the split, so that higher values lead to fewer splits
gamma - minimum loss reduction allowed for a split to occur
alpha - another name for L1 regularization
alpha - penalty on leaf weights rather than on feature weights
*alpha in linear or logistic regression is a penalty on feature weights
higher alpha values lead to more regularization
*this cause many leaf weights in the base learners to go to 0
lambda - another name for L2 regularization
lambda - a much smoother penalty than L1, causes leaf weights to smoothly decrease
instead of enforcing strong sparsity constraints on the leaf weights as in L1

example

```
import xgboost as xgb
import pandas as pd
boston_data = pd.read_csv("boston_data.csv")
X,y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]
boston_dmatrix = xgb.DMatrix(data=X,label=y)
params={"objective":"reg:linear","max_depth":4}
l1_params = [1,10,100]
rmses_l1=[]
for reg in l1_params:
    params["alpha"] = reg
    cv_results = xgb.cv(dtrain=boston_dmatrix, params=params,nfold=4,
                        num_boost_round=10,metrics="rmse",as_pandas=True,seed=123)
    rmses_l1.append(cv_results["test-rmse-mean"].tail(1).values[0])
print("Best rmse as a function of l1:")
print(pd.DataFrame(list(zip(l1_params,rmses_l1)), columns=["l1","rmse"]))
```

```
Best rmse as a function of l1:
    l1          rmse
0    1  69572.517742
1   10  73721.967141
2  100  82312.312413
```

#line 7 created a list of 3 different alpha or L1 values we want to try
#line 8 we initialize an empty list that will store our final root mean square error for each of these alpha values
#then we iterate our l1_params list through a for loop
#first creating a new key-value pair in our parameter dictionary that holds our

current alpha value
#then we run our XGBoost cross validation

Word on linear base learners
simply a sum of linear terms, exactly as you would find in a linear or logistic regression model
the ensemble itself will remain linear
*since you don't get any nonlinear combination of features in the final model, this approach is rarely used
*can get identical performance from a regularized linear model

This is why XGBoost is almost exclusively tree base learners
when the decision trees are all combined into an ensemble, their combination becomes a nonlinear function of each individual tree
making the ensemble itself nonlinear

Creating DataFrames from multiple equal-length lists
can use the list and zip function, one inside of the other, to convert multiple equal length lists into a single object that we can convert into a pandas DF
zip is a function that allows you to take multiple equal-length lists and iterate over them in parallel, side by side
*in Python 3, zip now creates a generator
a generator is an object that doesn't have to be completely instantiated at runtime
in order for the entire zipped pair of lists to be instantiated, we have to cast the zip generator object into a list directly
generators need to be completely instantiated before the can be used in DataFrame objects
list() instantiates the full generator and passing that into the DF converts the whole expression
example
pd.DataFrame(list(zip(list1, list2)), columns=['list1', 'list2']))
#zip creates a generator of parallel values
zip([1,2,3], ['a','b','c'])
output> [1,'a',], [2,'b'], [3,'c']

Example
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

reg_params = [1, 10, 100]

# Create the initial parameter dictionary for varying l2 strength: params
params = {"objective":"reg:linear","max_depth":3}

```
# Create an empty list for storing rmses as a function of l2 complexity
rmses_l2 = []

# Iterate over reg_params
for reg in reg_params:

    # Update l2 strength
    params["lambda"] = reg

    # Pass this updated param dictionary into cv
    cv_results_rmse = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=2,
num_boost_round=5, metrics="rmse", as_pandas=True, seed=123)

    # Append best rmse (final round) to rmses_l2
    rmses_l2.append(cv_results_rmse["test-rmse-mean"].tail(1).values[0])

# Look at best rmse per l2 param
print("Best rmse as a function of l2:")
print(pd.DataFrame(list(zip(reg_params, rmses_l2)), columns=["l2", "rmse"]))
```

Example

```
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params
params = {"objective":"reg:linear", "max_depth":2}

# Train the model: xg_reg
xg_reg = xgb.train(params=params, dtrain=housing_dmatrix,
num_boost_round=10)

# Plot the first tree
xgb.plot_tree(xg_reg, num_trees=0)
plt.show()

# Plot the fifth tree
xgb.plot_tree(xg_reg, num_trees=4)
plt.show()

# Plot the last tree sideways
xgb.plot_tree(xg_reg, num_trees=9, rankdir='LR')
plt.show()
```

with XGBoost can examine the importance of each feature column in the original dataset within the model

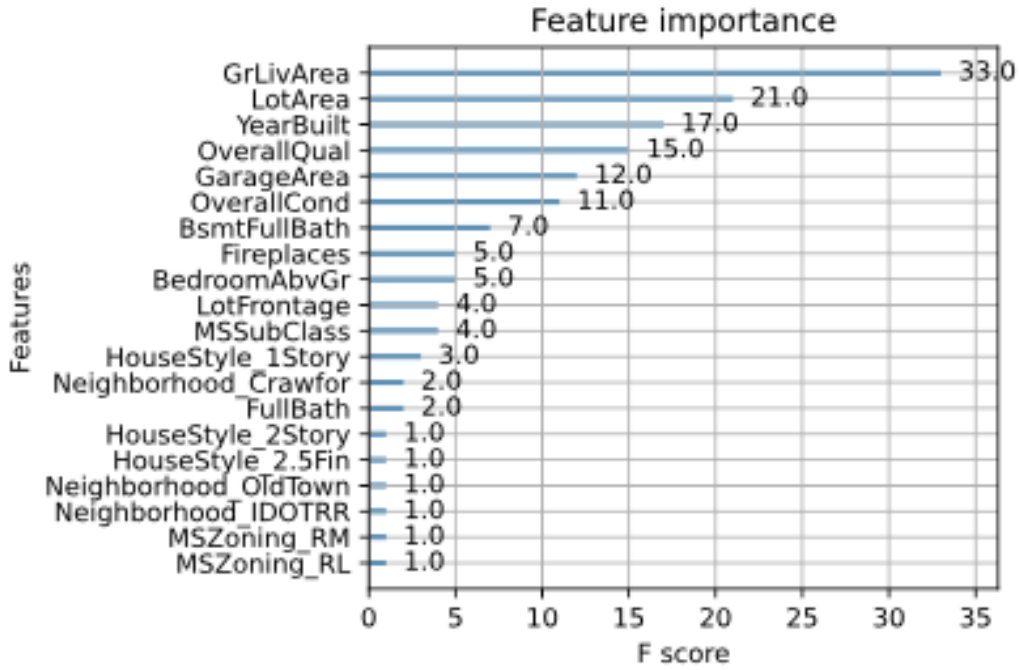
```
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary: params
params = {'objective':'reg:linear', 'max_depth':4}

# Train the model: xg_reg
xg_reg = xgb.train(params=params, dtrain=housing_dmatrix,
num_boost_round=10)

# Plot the feature importances
xgb.plot_importance(xg_reg)
plt.show()
```

output>



Why tune your model?
example
untuned version

```
import pandas as pd
import xgboost as xgb
import numpy as np
housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
X,y = housing_data[housing_data.columns.tolist()[:-1]],
        housing_data[housing_data.columns.tolist()[-1]]
housing_dmatrix = xgb.DMatrix(data=X,label=y)
untuned_params={"objective":"reg:linear"}
untuned_cv_results_rmse = xgb.cv(dtrain=housing_dmatrix,
        params=untuned_params,nfold=4,
        metrics="rmse",as_pandas=True,seed=123)
print("Untuned rmse: %f" %((untuned_cv_results_rmse["test-rmse-mean"]).tail(1)))
```

```
Untuned rmse: 34624.229980
```

when tuning we will build up a dictionary typically called a parameter grid
this can be seen in the below example

```
import pandas as pd
import xgboost as xgb
import numpy as np
housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
X,y = housing_data[housing_data.columns.tolist()[:-1]],
     housing_data[housing_data.columns.tolist()[-1]]
housing_dmatrix = xgb.DMatrix(data=X,label=y)
tuned_params = {"objective":"reg:linear",'colsample_bytree': 0.3,
    'learning_rate': 0.1, 'max_depth': 5}
tuned_cv_results_rmse = xgb.cv(dtrain=housing_dmatrix,
     params=tuned_params, nfold=4, num_boost_round=200, metrics="rmse",
     as_pandas=True, seed=123)
print("Tuned rmse: %f" %((tuned_cv_results_rmse["test-rmse-mean"]).tail(1)))
```

```
Tuned rmse: 29812.683594
```

*can see with tuning we got a 14% reduction in our RMSE
*goal is always the lowest RMSE possible
**reg:linear has deprecated in favor of reg:squarederror

Example
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

```
# Create the parameter dictionary for each tree: params
params = {"objective":"reg:linear", "max_depth":3}

# Create list of number of boosting rounds
num_rounds = [5, 10, 15]

# Empty list to store final round rmse per XGBoost model
final_rmse_per_round = []

# Iterate over num_rounds and build one model per num_boost_round parameter
for curr_num_rounds in num_rounds:

    # Perform cross-validation: cv_results
    cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=3,
num_boost_round=curr_num_rounds, metrics="rmse", as_pandas=True, seed=123)

    # Append final round RMSE
    final_rmse_per_round.append(cv_results["test-rmse-mean"].tail().values[-1])

# Print the resultant DataFrame
num_rounds_rmses = list(zip(num_rounds, final_rmse_per_round))
print(pd.DataFrame(num_rounds_rmses,columns=["num_boosting_rounds","rmse"]
))

output>
   num_boosting_rounds       rmse
    0                    5  50903.300
    1                   10  34774.194
    2                   15  32895.099
```

Early stopping
can be used with XGB models
tests the model after every boositing round against a holdout dataset, stopping training early if the holdout measure does not improve after a predetermined number of rounds
we will use rmse as our holdout measure
example

```
# Create your housing DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary for each tree: params
params = {"objective":"reg:linear", "max_depth":4}
```

```
# Perform cross-validation with early stopping: cv_results
cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=3,
early_stopping_rounds=10, num_boost_round=50, metrics='rmse',
as_pandas=True, seed=123)

# Print cv_results
print(cv_results)
```

XGBoost's hyperparameters
depend on type of base learner

for trees (most common)
learning rate - affects how quickly the model fits the residual error using additional base learners
low learning rate will require more boosting rounds to achieve the same reduction in residual error as an XGBoost mode with a high learning rate
gamma (described in earlier chapter) - minimize loss reduction to creat new tree split
lambda (described in earlier chapter) - L2 reg on leaf weights
alpha (described in earlier chapter) - L1 reg on leaf weights
max_depth - how deeply each tree is allowed to grow during each boosting round
subsample - percent of samples used per tree
subsample must be a value between 0 and 1 and is the fraction of the total training set that can be used for any given boosting round
a low value equates to a low fraction of you training data used per boosting round > this may lead to underfitting
a high value > may lead to overfitting
colsample_bytree - percent of features used per tree
the fraction of features used during any given boost round
using a small value can be considered additional regularization
using a large value in some cases can lead to overfitting

Sidebar - refresher on regularization
regularization acts as overfitting prevention
decreases the complexity of a model as it trains
this helps reduce the noise of a specific example
allows the model to generalize better
and hopefully be more effective on unseen data
L1 (Lasso) encourages some weights to become zero (which removes some features) > making certian other features more important
L2 (Ridge) penalty is proportional to the squares of the model's weights to the loss function > drives all the weights to smaller values
globally working to find a balance

the balance is the bias-variance trade-off
high reg reduces variance but increases bias
low reg reduces bias but increases variance
cross validation helps us to find this balance

Example
```
# Create your housing DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)

# Create the parameter dictionary for each tree (boosting round)
params = {"objective":"reg:linear", "max_depth":3}

# Create list of eta values and empty list to store final round rmse per xgboost model
eta_vals = [0.001, 0.01, 0.1]
best_rmse = []

# Systematically vary the eta
for curr_val in eta_vals:

    params["eta"] = curr_val

    # Perform cross-validation: cv_results
    cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=3, early_stopping_rounds=5, num_boost_round=10, metrics='rmse', as_pandas=True, seed=123)

    # Append the final round rmse to best_rmse
    best_rmse.append(cv_results["test-rmse-mean"].tail().values[-1])

# Print the resultant DataFrame
print(pd.DataFrame(list(zip(eta_vals, best_rmse)), columns=["eta","best_rmse"]))
```

Example
```
# Create your housing DMatrix
housing_dmatrix = xgb.DMatrix(data=X,label=y)

# Create the parameter dictionary
params = {"objective":"reg:linear"}

# Create list of max_depth values
max_depths = [2, 5, 10, 20]
best_rmse = []
```

```
# Systematically vary the max_depth
for curr_val in max_depths:

    params["max_depth"] = curr_val

    # Perform cross-validation
    cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=2,
early_stopping_rounds=5, num_boost_round=10, metrics='rmse', as_pandas=True,
seed=123)

    # Append the final round rmse to best_rmse
    best_rmse.append(cv_results["test-rmse-mean"].tail().values[-1])

# Print the resultant DataFrame
print(pd.DataFrame(list(zip(max_depths,
best_rmse)),columns=["max_depth","best_rmse"]))

output>
   max_depth  best_rmse
    0          2  37957.469
    1          5  35596.600
    2         10  36065.547
    3         20  36739.576

Example
# Create your housing DMatrix
housing_dmatrix = xgb.DMatrix(data=X,label=y)

# Create the parameter dictionary
params={"objective":"reg:linear","max_depth":3}

# Create list of hyperparameter values: colsample_bytree_vals
colsample_bytree_vals = [0.1, 0.5, 0.8, 1]
best_rmse = []

# Systematically vary the hyperparameter value
for curr_val in colsample_bytree_vals:

    params['colsample_bytree'] = curr_val

    # Perform cross-validation
    cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=2,
```

```
            num_boost_round=10, early_stopping_rounds=5,
            metrics="rmse", as_pandas=True, seed=123)

    # Append the final round rmse to best_rmse
    best_rmse.append(cv_results["test-rmse-mean"].tail().values[-1])

# Print the resultant DataFrame
print(pd.DataFrame(list(zip(colsample_bytree_vals, best_rmse)),
columns=["colsample_bytree","best_rmse"]))

output>
  colsample_bytree  best_rmse
    0              0.1  50033.735
    1              0.5  35656.186
    2              0.8  36399.002
    3              1.0  35836.044
```

Grid search and random search
*how to find optimal values for several hyperparameters simultaneously
this can be challenging when they interact in non-obvious, non-linear ways

Review of Grid Search
a method of exhaustively searching through a collection of possible parameter values
*searches once per set of hyper parameters
number of models = number of distinct values per hyperparameter multiplied across each hyperparameter
pick the parameter configuration that gave you the best value for the metric (example rmse) you were using
example

```
import pandas as pd
import xgboost as xgb
import numpy as np
from sklearn.model_selection import GridSearchCV
housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
X, y = housing_data[housing_data.columns.tolist()[:-1]],
       housing_data[housing_data.columns.tolist()[-1]
housing_dmatrix = xgb.DMatrix(data=X,label=y)
gbm_param_grid = {'learning_rate': [0.01,0.1,0.5,0.9],
                  'n_estimators': [200],
                  'subsample': [0.3, 0.5, 0.9]}
gbm = xgb.XGBRegressor()
grid_mse = GridSearchCV(estimator=gbm,param_grid=gbm_param_grid,
            scoring='neg_mean_squared_error', cv=4, verbose=1)
grid_mse.fit(X, y)
print("Best parameters found: ",grid_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(grid_mse.best_score_)))
```

```
Best parameters found: {'learning_rate': 0.1,
'n_estimators': 200, 'subsample': 0.5}
Lowest RMSE found:  28530.1829341
```

Random search
you decide how many models, or iterations, you want to try out before stopping
draws a random combination of possible hyperparameter values from the range of allowable hyperparameters a set number of times
once you have created the number of models you had specified initially, you simply pick the best one
*just side reminder learning rate is also called eta
example

```
import pandas as pd
import xgboost as xgb
import numpy as np
from sklearn.model_selection import RandomizedSearchCV
housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
X,y = housing_data[housing_data.columns.tolist()[:-1]],
      housing_data[housing_data.columns.tolist()[-1]]
housing_dmatrix = xgb.DMatrix(data=X,label=y)
gbm_param_grid = {'learning_rate': np.arange(0.05,1.05,.05),
                  'n_estimators': [200],
                  'subsample': np.arange(0.05,1.05,.05)}
gbm = xgb.XGBRegressor()
randomized_mse = RandomizedSearchCV(estimator=gbm, param_distributions=gbm_param_grid,
                        n_iter=25, scoring='neg_mean_squared_error', cv=4, verbose=1)
randomized_mse.fit(X, y)
print("Best parameters found: ",randomized_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(randomized_mse.best_score_)))
```

```
Best parameters found: {'subsample': 0.60000000000000009,
'n_estimators': 200, 'learning_rate': 0.20000000000000001}
Lowest RMSE found: 28300.2374291
```

Example

```
# Create the parameter grid: gbm_param_grid
gbm_param_grid = {
   'colsample_bytree': [0.3, 0.7],
   'n_estimators': [50],
   'max_depth': [2, 5]
}

# Instantiate the regressor: gbm
gbm = xgb.XGBRegressor()

# Perform grid search: grid_mse
grid_mse = GridSearchCV(estimator=gbm, param_grid=gbm_param_grid,
scoring='neg_mean_squared_error', cv=4, verbose=1)


# Fit grid_mse to the data
grid_mse.fit(X, y)

# Print the best parameters and lowest RMSE
```

```
print("Best parameters found: ", grid_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(grid_mse.best_score_)))

output>
Fitting 4 folds for each of 4 candidates, totalling 16 fits
   Best parameters found:  {'colsample_bytree': 0.3, 'max_depth': 5, 'n_estimators':
50}
   Lowest RMSE found:  29916.017850830365

# Create the parameter grid: gbm_param_grid
gbm_param_grid = {
   'n_estimators': [25],
   'max_depth': np.arange(2, 11)
}

# Instantiate the regressor: gbm
gbm = xgb.XGBRegressor(n_estimators=10)

# Perform random search: grid_mse
randomized_mse = RandomizedSearchCV(estimator=gbm,
param_distributions=gbm_param_grid, scoring='neg_mean_squared_error',
n_iter=5, cv=4, verbose=1)


# Fit randomized_mse to the data
randomized_mse.fit(X, y)

# Print the best parameters and lowest RMSE
print("Best parameters found: ", randomized_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(randomized_mse.best_score_)))

output>
Fitting 4 folds for each of 5 candidates, totalling 20 fits
   Best parameters found:  {'n_estimators': 25, 'max_depth': 5}
   Lowest RMSE found:  31043.162060428804
```

Limitations of Grid Search
time and efficiency
can become a serious issue as the amount of distinct values and hyperparameters increases

Limitations of Random Search
the parameter space can become massive

randomly searching through this space can leave you hoping for just a good result, little own the best result

Pipeline review
pipelines in sklearn are objects that take a list of named tuples as input
the named tuples must always contain a string name as the first element in each tuple
than any scikit-learn compatible transformer or estimator object as the second element
each named tuple in the pipeline is called a step
the list of transformations that are contained in the list are executed in order once some data is passed through the pipeline
this is done using standard fit/predict paradigm
**where pipelines are really useful is that they can be used as input estimator objects into other scikit objects themselves
most useful is the cross_val_score method
this allows for efficien cross-validation and out of sample metric calculation
along with grid search and random search approaches for tuning hyperparameters
example

```
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
names = ["crime","zone","industry","charles","no","rooms",
        "age", "distance","radial","tax","pupil","aam","lower","med_price"]

data = pd.read_csv("boston_housing.csv",names=names)

X, y = data.iloc[:,:-1], data.iloc[:,-1]
rf_pipeline = Pipeline[("st_scaler",
                StandardScaler()),
                ("rf_model",RandomForestRegressor())]

scores = cross_val_score(rf_pipeline,X,y,
scoring="neg_mean_squared_error",cv=10)
```

*side - neg_mean_squared_error is scikit's API specific way of calculating the mean squared error
negative mean squared errors don't exist
all squares must be positive when working with real numbers

we finish the above off this way
final_avg_rmse = np.mean(np.sqrt(np.abs(scores)))
print(‘Final RMSE:’, final_avg_rmse)

Further preprocessing may be needed depending on the complexity of the dataset
first approach
use the LabelEncoder and OneHotEncoder
LabelEncoder converts a categorical column of strings into integers that map onto those strings
OneHotEncoder takes a column of intergers that are treated as categorical values and encodes them as dummy variables
this approach cannot be done within the pipeline
second approach
DictVectorizer
a class found in scikit feature extraction submodule
started use in text processing pipelines by converting lists of feature mappings into vectors
need to convert our DataFrame into a list of dictionary entries
this can accomplish both above steps in one line of code

```
# Import LabelEncoder
from sklearn.preprocessing import LabelEncoder

# Fill missing values with 0
df.LotFrontage = df['LotFrontage'].fillna(0)

# Create a boolean mask for categorical columns
categorical_mask = (df.dtypes == object)

# Get list of categorical column names
categorical_columns = df.columns[categorical_mask].tolist()

# Print the head of the categorical columns
print(df[categorical_columns].head())

# Create LabelEncoder object: le
le = LabelEncoder()

# Apply LabelEncoder to categorical columns
df[categorical_columns] = df[categorical_columns].apply(lambda x:
le.fit_transform(x))

# Print the head of the LabelEncoded categorical columns
```

```
print(df[categorical_columns].head())

# Import OneHotEncoder
from sklearn.preprocessing import OneHotEncoder

# Create OneHotEncoder: ohe
ohe = OneHotEncoder(sparse=False)

# Apply OneHotEncoder to categorical columns - output is no longer a dataframe:
df_encoded
df_encoded = ohe.fit_transform(df)

# Print first 5 rows of the resulting dataset - again, this will no longer be a pandas
dataframe
print(df_encoded[:5, :])

# Print the shape of the original DataFrame
print(df.shape)

# Print the shape of the transformed array
print(df_encoded.shape)

or just do this
# Import DictVectorizer
from sklearn.feature_extraction import DictVectorizer

# Convert df into a dictionary: df_dict
df_dict = df.to_dict('records')

# Create the DictVectorizer object: dv
dv = DictVectorizer(sparse=False)

# Apply dv on df: df_encoded
df_encoded = dv.fit_transform(df_dict)

# Print the resulting first five rows
print(df_encoded[:5,:])

# Print the vocabulary
print(dv.vocabulary_)

Example
# Import necessary modules
```

```
from sklearn.feature_extraction import DictVectorizer
from sklearn.pipeline import Pipeline

# Fill LotFrontage missing values with 0
X.LotFrontage = X.LotFrontage.fillna(0)

# Setup the pipeline steps: steps
steps = [("ohe_onestep", DictVectorizer(sparse=False)),
        ("xgb_model", xgb.XGBRegressor())]

# Create the pipeline: xgb_pipeline
xgb_pipeline = Pipeline(steps)

# Fit the pipeline
xgb_pipeline.fit(X.to_dict('records'),y)
```

Scikit pipeline example with XGBoost

```
import pandas as pd
import xgboost as xgb
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
names = ["crime","zone","industry","charles","no","rooms","age",
        "distance","radial","tax","pupil","aam","lower","med_price"]
data = pd.read_csv("boston_housing.csv",names=names)
X, y = data.iloc[:,:-1], data.iloc[:,-1]
xgb_pipeline = Pipeline[("st_scaler", StandardScaler()),
                        ("xgb_model",xgb.XGBRegressor())]
scores = cross_val_score(xgb_pipeline, X, y,
                         scoring="neg_mean_squared_error",cv=10)
final_avg_rmse = np.mean(np.sqrt(np.abs(scores)))
print("Final XGB RMSE:", final_avg_rmse)
```

```
Final RMSE: 4.02719593323
```

Sometime complex wrangling must be done in order to use XGBoost within a sklearn pipeline
the next example will show this
sklearn and pandas do not always communicate with each other appropriately

as sklearn objects uses np arrays and pandas uses DataFrames
to bridge this gap we use sklearn_pandas
this libraray has a special class called DataFrameMapper
allows for easy conversion between NumPy arrays and pandas DataFrames
we will also use an uncommon aspect of sklearn the sklearn.impute import SimpleImputer
an impute submodule
allows us to fill in missing numerical and categorical values
also within the sklearn.pipeline we will use FeatureUnion class
a pipeline submodule
allows us to combine separate pipeline outputs into a single pipeline output
why?
what we would need to do if we had one set of preprocessing steps we needed to perform on the categorical features of a dataset and a distinct set of preprocessing steps on the numeric features found in a dataset
another way of saying this is combining multiple pipelines of features into a single pipeline of features

Example

```
# Import necessary modules
from sklearn.feature_extraction import DictVectorizer
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

# Fill LotFrontage missing values with 0
X.LotFrontage = X.LotFrontage.fillna(0)

# Setup the pipeline steps: steps
steps = [("ohe_onestep", DictVectorizer(sparse=False)),
        ("xgb_model", xgb.XGBRegressor(max_depth=2, objective="reg:linear"))]

# Create the pipeline: xgb_pipeline
xgb_pipeline = Pipeline(steps)

# Cross-validate the model
cross_val_scores = cross_val_score(xgb_pipeline, X.to_dict('records'), y, cv=10,
scoring='neg_mean_squared_error')

# Print the 10-fold RMSE
print("10-fold RMSE: ", np.mean(np.sqrt(np.abs(cross_val_scores))))
```

Example

```
# Import necessary modules
```

```
from sklearn_pandas import DataFrameMapper
from sklearn.impute import SimpleImputer

# Check number of nulls in each feature column
nulls_per_column = X.isnull().sum()
print(nulls_per_column)

# Create a boolean mask for categorical columns
categorical_feature_mask = X.dtypes == object

# Get list of categorical column names
categorical_columns = X.columns[categorical_feature_mask].tolist()

# Get list of non-categorical column names
non_categorical_columns = X.columns[~categorical_feature_mask].tolist()

# Apply numeric imputer
numeric_imputation_mapper = DataFrameMapper(
                                [([numeric_feature], SimpleImputer(strategy="median"))
for numeric_feature in non_categorical_columns],
                                input_df=True,
                                df_out=True
                               )

# Apply categorical imputer
categorical_imputation_mapper = DataFrameMapper(
                                    [(category_feature, SimpleImputer()) for
category_feature in categorical_columns],
                                    input_df=True,
                                    df_out=True
                                   )



Using FeatureUnion
# Import FeatureUnion
from sklearn.pipeline import FeatureUnion

# Combine the numeric and categorical transformations
numeric_categorical_union = FeatureUnion([
                                ("num_mapper", numeric_imputation_mapper),
                                ("cat_mapper", categorical_imputation_mapper)
                               ])
```

Put it all together
# Create full pipeline
pipeline = Pipeline([
("featureunion", numeric_categorical_union),
("dictifier", Dictifier()),
("vectorizer", DictVectorizer(sort=False)),
("clf", xgb.XGBClassifier(max_depth=3))
])

# Perform cross-validation
cross_val_scores = cross_val_score(pipeline, kidney_data, y, scoring="roc_auc", cv=3)

# Print avg. AUC
print("3-fold AUC: ", np.mean(cross_val_scores))

Tuning XGBoost hyperparameters in a pipeline
example

```
import pandas as pd
   ...: import xgboost as xgb
   ...: import numpy as np
   ...: from sklearn.preprocessing import StandardScaler
   ...: from sklearn.pipeline import Pipeline
   ...: from sklearn.model_selection import RandomizedSearchCV
names = ["crime","zone","industry","charles","no",
   ...: "rooms","age", "distance","radial","tax",
   ...: "pupil","aam","lower","med_price"]
data = pd.read_csv("boston_housing.csv",names=names)
X, y = data.iloc[:,:-1],data.iloc[:,-1]
xgb_pipeline = Pipeline[("st_scaler",
   ...: StandardScaler()), ("xgb_model",xgb.XGBRegressor())]
gbm_param_grid = {
   ...:     'xgb_model__subsample': np.arange(.05, 1, .05),
   ...:     'xgb_model__max_depth': np.arange(3,20,1),
   ...:     'xgb_model__colsample_bytree': np.arange(.1,1.05,.05) }
randomized_neg_mse = RandomizedSearchCV(estimator=xgb_pipeline,
   ...: param_distributions=gbm_param_grid, n_iter=10,
   ...: scoring='neg_mean_squared_error', cv=4)
randomized_neg_mse.fit(X, y)
```

**main difference
in order for the hyperparmeters to be passed to the appropriate step, you have to

name the parameters in the dictionary with the name of the step being referenced followed by 2 underscore signs
in this example it is xgb_model__

Example
```
# Create the parameter grid
gbm_param_grid = {
    'clf__learning_rate': np.arange(0.05, 1, 0.05),
    'clf__max_depth': np.arange(3, 10, 1),
    'clf__n_estimators': np.arange(50, 200, 50)
}

# Perform RandomizedSearchCV
randomized_roc_auc = RandomizedSearchCV(estimator=pipeline,
param_distributions=gbm_param_grid, n_iter=2, scoring='roc_auc',cv=2,
verbose=1)

# Fit the estimator
randomized_roc_auc.fit(X,y)

# Compute metrics
print(randomized_roc_auc.best_score_)
print(randomized_roc_auc.best_estimator_)
```

**can also use XGB for ranking and recommendation
**powerful tool using hyperparameter tuning with Bayesian Optimiazation
**lastly, using XGB as part of an ensemble of other models for regression/ classification