Extreme Gradient Boosting with XGBoost

20 minute read

XGBoost: Fit/Predict

It's time to create our first XGBoost model! We can use the scikit-learn .fit() / .predict() paradigm that we are already familiar to build your XGBoost models, as the xgboost library has a scikit-learn compatible API! Here, we'll be working with churn data. This dataset contains imaginary data from a ride-sharing app with user behaviors over their first month of app usage in a set of imaginary cities as well as whether they used the service 5 months after sign-up.

Our goal is to use the first month's worth of data to predict whether the app's users will remain users of the service at the 5 month mark. This is a typical setup for a churn prediction problem. To do this, we'll split the data into training and test sets, fit a small xgboost model on the training set, and evaluate its performance on the test set by computing its accuracy.

pandas and numpy have been imported as pd and np, and train_test_split has been imported from sklearn.model_selection. Additionally, the arrays for the features and the target have been created as X and y.

```python
#import xgboost
import xgboost as xgb
# Create arrays for the features and the target: X, y
X, y = churn_data.iloc[:,:-1], churn_data.iloc[:,-1]
# 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 XGBClassifier: xg_cl
xg_cl = xgb.XGBClassifier(objective='binary:logistic', n_estimators=10, seed=123)
# Fit the classifier to the training set
xg_cl.fit(X_train,y_train)
# Predict the labels of the test set: preds
preds = xg_cl.predict(X_test)
# Compute the accuracy: accuracy
accuracy = float(np.sum(preds==y_test))/y_test.shape[0]
print("accuracy: %f" % (accuracy))
```

Decision trees

Our task in this exercise is to make a simple decision tree using scikit-learn's DecisionTreeClassifier on the breast cancer dataset that comes pre-loaded with scikit-learn. This dataset contains numeric measurements of various dimensions of individual tumors (such as perimeter and texture) from breast biopsies and a single outcome value (the tumor is either malignant, or benign). We've preloaded the dataset of samples (measurements) into X and the target values per tumor into y. Now, you have to split the complete dataset into training and testing sets, and then train a DecisionTreeClassifier. You'll specify a parameter called max_depth. Many other parameters can be modified within this model, and we can check all of them out here (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier).

```python
# 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)
```

Measuring accuracy

We'll now practice using XGBoost's learning API through its baked in cross-validation capabilities. XGBoost gets its lauded performance and efficiency gains by utilizing its own optimized data structure for datasets called a DMatrix. In the previous exercise, the input datasets were converted into DMatrix data on the fly, but when we use the xgboost cv object, we have to first explicitly convert your data into a DMatrix. So, that's what we will do here before running cross-validation on churn_data.

```
# Create the DMatrix: churn_dmatrix
churn_dmatrix = xgb.DMatrix(data=churn_data.iloc[:,:-1], label=churn_data.iloc[:,-1])
# 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]))
```

Measuring AUC

Now that we've used cross-validation to compute average out-of-sample accuracy (after converting from an error), it's very easy to compute any other metric you might be interested in. All we have to do is pass it (or a list of metrics) in as an argument to the metrics parameter of xgb.cv(). Our job in this exercise is to compute another common metric used in binary classification - the area under the curve ("auc").

```
# 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])
```

Decision trees as base learners

It's now time to build an XGBoost model to predict house prices - not in Boston, Massachusetts, but in Ames, Iowa! This dataset of housing prices has been pre-loaded into a DataFrame called df. If we explore it in the Shell, we'll see that there are a variety of features about the house and its location in the city. Our goal is to use trees as base learners. By default, XGBoost uses trees as base learners, so we don't have to specify that you want to use trees here with booster="gbtree". xgboost has been imported as xgb and the arrays for the features and the target are available in X and y, respectively.

```python
# 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 XGBRegressor: xg_reg
xg_reg = xgb.XGBRegressor(objective='reg:linear', n_estimators=10, seed=123)
# Fit the regressor to the training set
xg_reg.fit(X_train, y_train)
# Predict the labels of the test set: preds
preds = xg_reg.predict(X_test)
# Compute the rmse: rmse
rmse = np.sqrt(mean_squared_error(y_test, preds))
print("RMSE: %f" % (rmse))
```

Linear base learners

Now that we've used trees as base models in XGBoost, let's use the other kind of base model that can be used with XGBoost - a linear learner. This model, although not as commonly used in XGBoost, allows us to create a regularized linear regression using XGBoost's powerful learning API. However, because it's uncommon, we have to use XGBoost's own non-scikit-learn compatible functions to build the model, such as xgb.train(). In order to do this you must create the parameter dictionary that describes the kind of booster you want to use. The key-value pair that defines the booster type (base model) you need is "booster":"gblinear". Once you've created the model, you can use the .train() and .predict() methods of the model just like you've done in the past. Here, the data has already been split into training and testing sets, so we can dive right into creating the DMatrix objects required by the XGBoost learning API.

```python
# 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))
```

Evaluating model quality

It’s now time to begin evaluating model quality. Here, we will compare the RMSE and MAE of a cross-validated XGBoost model on the Ames housing data.

```python
# 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="rmse", as_pandas=True, seed=123)
# Print cv_results
print(cv_results)
# Extract and print final boosting round metric
print((cv_results["test-rmse-mean"]).tail(1))
```

Using regularization in XGBoost

We will now vary the l2 regularization penalty - also known as “lambda” - and see its effect on overall model performance on the Ames housing dataset.

```python
# 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"]))
```

Visualizing individual XGBoost trees

Now that we’ve used XGBoost to both build and evaluate regression as well as classification models, we should get a handle on how to visually explore your models. Here, we will visualize individual trees from the fully boosted model that XGBoost creates using the entire housing dataset. XGBoost has a plot_tree() function that makes this type of visualization easy. Once we train a model using the XGBoost learning API, we can pass it to the plot_tree() function along with the number of trees you want to plot using the num_trees argument.

```
# 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()
```

Visualizing feature importances: What features are most important in my dataset

Another way to visualize our XGBoost models is to examine the importance of each feature column in the original dataset within the model. One simple way of doing this involves counting the number of times each feature is split on across all boosting rounds (trees) in the model, and then visualizing the result as a bar graph, with the features ordered according to how many times they appear. XGBoost has a plot_importance() function that allows us to do exactly this, and we'll get a chance to use it in this exercise!

```
# 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()
```

Tuning the number of boosting rounds

Let's start with parameter tuning by seeing how the number of boosting rounds (number of trees you build) impacts the out-of-sample performance of our XGBoost model. We'll use xgb.cv() inside a for loop and build one model per num_boost_round parameter. Here, we'll continue working with the Ames housing dataset. The features are available in the array X, and the target vector is contained in y.

```
# 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"]))
```

Automated boosting round selection using early_stopping

Now, instead of attempting to cherry pick the best possible number of boosting rounds, we can very easily have XGBoost automatically select the number of boosting rounds for us within xgb.cv(). This is done using a technique called early stopping.

Early stopping works by testing the XGBoost model after every boosting round against a hold-out dataset and stopping the creation of additional boosting rounds (thereby finishing training of the model early) if the hold-out metric ("rmse" in our case) does not improve for a given number of rounds. Here we will use the early_stopping_rounds parameter in xgb.cv() with a large possible number of boosting rounds (50). Bear in mind that if the holdout metric continuously improves up through when num_boosting_rounds is reached, then early stopping does not occur. Here, the DMatrix and parameter dictionary have been created for us. Our task is to use cross-validation with early stopping. Go for it!

```
# 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)
```

Tuning eta

It's time to practice tuning other XGBoost hyperparameters in earnest and observing their effect on model performance! We'll begin by tuning the "eta", also known as the learning rate. The learning rate in XGBoost is a parameter that can range between 0 and 1, with higher values of "eta" penalizing feature weights more strongly, causing much stronger regularization.

```
# 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
    # 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["eta"] = curr_val

    # Perform cross-validation
    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(max_depths, best_rmse)),columns=["max_depth","best_rmse"]))
    # 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"]))
```

Tuning max_depth

Our job is to tune max_depth, which is the parameter that dictates the maximum depth that each tree in a boosting round can grow to. Smaller values will lead to shallower trees, and larger values to deeper trees.

```
# 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["eta"] = curr_val
    # Perform cross-validation
    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(max_depths, best_rmse)),columns=["max_depth","best_rmse"]))
```

Tuning colsample_bytree

Now, it's time to tune "colsample_bytree". We've already seen this if we've ever worked with scikit-learn's RandomForestClassifier or RandomForestRegressor, where it just was called max_features. In both xgboost and sklearn, this parameter (although named differently) simply specifies the fraction of features to choose from at every split in a given tree. In xgboost, colsample_bytree must be specified as a float between 0 and 1.

```python
# 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["eta"] = 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"]))
```

Grid Search with XGBoost

Let’s take our parameter tuning to the next level by using scikit-learn’s GridSearch and RandomizedSearch capabilities with internal cross-validation using the GridSearchCV and RandomizedSearchCV functions. We will use these to find the best model exhaustively from a collection of possible parameter values across multiple parameters simultaneously. Let’s get to work, starting with GridSearchCV!

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

Random Search with XGBoost

Often, GridSearchCV can be really time consuming, so in practice, we may want to use RandomizedSearchCV instead, as we will do in this exercise. The good news is we only have to make a few modifications to your GridSearchCV code to do RandomizedSearchCV. The key difference is we have to specify a param_distributions parameter instead of a param_grid parameter.

```python
# Create the parameter grid: gbm_param_grid
gbm_param_grid = {
    'n_estimators': [25],
    'max_depth': range(2, 12)
}
# Instantiate the regressor: gbm
gbm = xgb.XGBRegressor(n_estimators=10)
# Perform random search: grid_mse
randomized_mse = RandomizedSearchCV(param_distributions=gbm_param_grid, estimator=gbm,
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_)))
```

Encoding categorical columns I: LabelEncoder

Now that we’ve seen what will need to be done to get the housing data ready for XGBoost, let’s go through the process step-by-step. First, we will need to fill in missing values - as we saw previously, the column LotFrontage has many missing values. Then, we will need to encode any categorical columns in the dataset using one-hot encoding so that they are encoded numerically. We can watch this video from Supervised Learning with scikit-learn for a refresher on the idea.

The data has five categorical columns: MSZoning, PavedDrive, Neighborhood, BldgType, and HouseStyle. Scikit-learn has a LabelEncoder (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html) function that converts the values in each categorical column into integers.

```
# 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())
```

Encoding categorical columns II: OneHotEncoder

Okay - so we have our categorical columns encoded numerically. Can you now move onto using pipelines and XGBoost? Not yet! In the categorical columns of this dataset, there is no natural ordering between the entries. As an example: Using LabelEncoder, the CollgCr Neighborhood was encoded as 5, while the Veenker Neighborhood was encoded as 24, and Crawfor as 6. Is Veenker "greater" than Crawfor and CollgCr? No - and allowing the model to assume this natural ordering may result in poor performance. As a result, there is another step needed: We have to apply a one-hot encoding to create binary, or "dummy" variables. We can do this using scikit-learn's OneHotEncoder (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html).

```
# Import OneHotEncoder
from sklearn.preprocessing import OneHotEncoder
# Create OneHotEncoder: ohe
ohe = OneHotEncoder(categorical_features=categorical_mask, 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)
```

Encoding categorical columns III: DictVectorizer

Alright, one final trick before we dive into pipelines. The two step process you just went through - LabelEncoder followed by OneHotEncoder - can be simplified by using a DictVectorizer. Using a DictVectorizer on a DataFrame that has been converted to a dictionary allows us to get label encoding as well as one-hot encoding in one go. Our task is to work through this strategy in this exercise!

```
# 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_)
```

Preprocessing within a pipeline

Now that we've seen what steps need to be taken individually to properly process the Ames housing data, let's use the much cleaner and more succinct DictVectorizer approach and put it alongside an XGBoostRegressor inside of a scikit-learn pipeline.

```
# 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)
```

Cross-validating your XGBoost model

In this exercise, we'll go one step further by using the pipeline you've created to preprocess and cross-validate our model.

```
# 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))))
```

Kidney disease case study I: Categorical Imputer

We'll now continue your exploration of using pipelines with a dataset that requires significantly more wrangling. The chronic kidney disease dataset (https://archive.ics.uci.edu/ml/datasets/chronic_kidney_disease) contains both categorical and numeric features, but contains lots of missing values. The goal here is to predict who has chronic kidney disease given various blood indicators as features.

We will introduce a new library, sklearn_pandas (https://github.com/pandas-dev/sklearn-pandas), that allows us to chain many more processing steps inside of a pipeline than are currently supported in scikit-learn. Specifically, we'll be able to impute missing categorical values directly using the Categorical_Imputer() class in sklearn_pandas, and the DataFrameMapper() class to apply any arbitrary sklearn-compatible transformer on DataFrame columns, where the resulting output can be either a NumPy array or DataFrame.

We've also created a transformer called a Dictifier that encapsulates converting a DataFrame using .to_dict("records") without you having to do it explicitly (and so that it works in a pipeline). Finally, we've also provided the list of feature names in kidney_feature_names, the target name in kidney_target_name, the features in X, and the target in y.

Our task is to apply the CategoricalImputer to impute all of the categorical columns in the dataset. We can refer to how the numeric imputation mapper was created as a template. Notice the keyword arguments input_df=True and df_out=True? This is so that you can work with DataFrames instead of arrays. By default, the transformers are passed a numpy array of the selected columns as input, and as a result, the output of the DataFrame mapper is also an array. Scikit-learn transformers have historically been designed to work with numpy arrays, not pandas DataFrames, even though their basic indexing interfaces are similar.

```
# Import necessary modules
from sklearn_pandas import DataFrameMapper
from sklearn_pandas import CategoricalImputer
# 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],Imputer(strategy="median"))
for numeric_feature in non_categorical_columns],
                                            input_df=True,
                                            df_out=True
                                           )
# Apply categorical imputer
categorical_imputation_mapper = DataFrameMapper(
                                                [(category_feature, CategoricalImputer())
for category_feature in categorical_columns],
                                                input_df=True,
                                                df_out=True
                                               )
```

Kidney disease case study II: Feature Union

Having separately imputed numeric as well as categorical columns, our task is now to use scikit-learn's FeatureUnion (http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.FeatureUnion.html) to concatenate their results, which are contained in two separate transformer objects - numeric_imputation_mapper, and categorical_imputation_mapper, respectively. Just like with pipelines, we have to pass it a list of (string, transformer) tuples, where the first half of each tuple is the name of the transformer.

```python
# 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)
                                         ])
```

Kidney disease case study III: Full pipeline

It's time to piece together all of the transforms along with an XGBClassifier to build the full pipeline! Besides the numeric_categorical_union that we created in the previous exercise, there are two other transforms needed: the Dictifier() transform which we created for you, and the DictVectorizer(). After creating the pipeline, our task is to cross-validate it to see how well it performs.

```python
# 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))
```

Bringing it all together

Alright, it's time to bring together everything we've learned so far! In this final exercise of the course, we will combine our work from the previous exercises into one end-to-end XGBoost pipeline to really cement our understanding of preprocessing and pipelines in XGBoost. Our job is to perform a randomized search and identify the best hyperparameters.

```python
# Create the parameter grid
gbm_param_grid = {
    'clf__learning_rate': np.arange(.05, 1, .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_)
```

Datasets:

- Ames housing prices (preprocessed) (https://assets.datacamp.com/production/repositories/943/datasets/4dbcaee889ef06fb0763e4a8652a4c1f268359b2/ames_housing_trimmed_processed.csv)
- Ames housing prices (original) (https://assets.datacamp.com/production/course_6611/datasets/ames_unprocessed_data.csv)
- Chronic kidney disease (https://assets.datacamp.com/production/course_6611/datasets/chronic_kidney_disease.csv)

Tags: python

Updated: December 03, 2018