Lec.3. XGBoost

Machine Learning II

Aidos Sarsembayev, IITU, Almaty, 2019

Outline

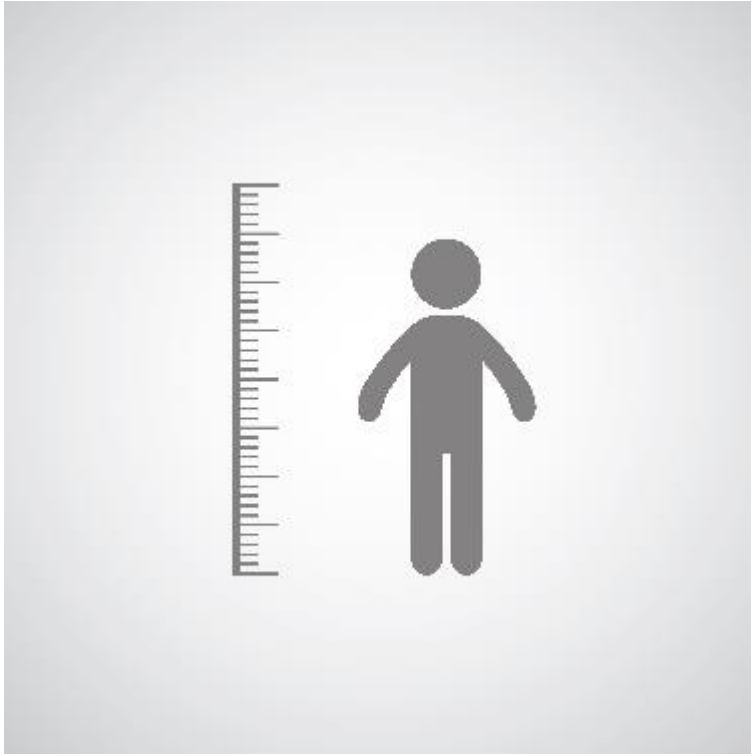
1. Regression with XGB
   - Regression basics
   - Common regression metrics
   - Common regression algorithms
   - Objective (loss) functions and base learners
   - Regularization and base learners in XGBoost
2. Fine-tuning your XGBoost model
   - Why tune your model?
   - Tunable parameters in XGBoost
   - Review of Grid Search and Random Search
   - Limits of Grid Search and Random Search

Regression with XGB

Regression with XGB

Regression basics

Outcome is real-valued



Common regression metrics

Common regression metrics

- Root mean squared error (RMSE)
- Mean absolute error (MAE)

Computing RMSE

Actual	Predicted
10	20
3	8
6	1

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(f_i - o_i)^2}$$

Computing RMSE

Actual	Predicted	Error
10	20	-10
3	8	-5
6	1	5

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(f_i - o_i)^2}$$

Computing RMSE

Actual	Predicted	Error	RMSE
10	20	-10	100
3	8	-5	25
6	1	5	25

Total Squared Error: 150
Mean Squared Error: 50
Root Mean Squared Error: 7.07

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(f_i - o_i)^2}$$

Computing MAE

Actual	Predicted	Error
10	20	-10
3	8	-5
6	1	5

Total Absolute Error: 20
Mean Absolute Error: 6.67

Common regression algorithms

- Linear regression
- Decision trees

Common regression algorithms

- Linear regression
- Decision trees

  we know that DT are both for regression and classification

Objective (loss) functions and base learners

Objective Functions and Why We Use Them

- Quantifies how far off a prediction is from the actual result
- Measures the difference between estimated and true values for some collection of data
- Goal: Find the model that yields the minimum value of the loss function

Common Loss Functions and XGBoost

- Loss function names in xgboost:
  - reg:linear - use for regression problems
  - reg:logistic - use for classification problems when you want just decision, not probability
  - binary:logistic - use when you want probability rather than just decision

Base Learners and Why We Need Them

- XGBoost involves creating a meta-model that is composed of many individual models that combine to give a final prediction
- Individual models = base learners
- Want base learners that when combined create final prediction that is non-linear
- Each base learner should be good at distinguishing or predicting different parts of the dataset
- Two kinds of base learners: tree and linear

Trees as Base Learners example: Scikit-learn API

```
In [1]: import xgboost as xgb

In [2]: import pandas as pd

In [3]: import numpy as np

In [4]: from sklearn.model_selection import train_test_split

In [5]: boston_data = pd.read_csv("boston_housing.csv")

In [6]: X, y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]

In [7]: X_train, X_test, y_train, y_test= train_test_split(X, y,
        test_size=0.2, random_state=123)

In [8]: xg_reg = xgb.XGBRegressor(objective='reg:linear',
        n_estimators=10, seed=123)

In [9]: xg_reg.fit(X_train, y_train)

In [10]: preds = xg_reg.predict(X_test)
In [11]: rmse = np.sqrt(mean_squared_error(y_test,preds))

In [12]: print("RMSE: %f" % (rmse))
RMSE: 129043.2314
```

Linear Base Learners Example:

```
In [1]: import xgboost as xgb

In [2]: import pandas as pd

In [3]: import numpy as np

In [4]: from sklearn.model_selection import train_test_split

In [5]: boston_data = pd.read_csv("boston_housing.csv")

In [6]: X, y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]

In [7]: X_train, X_test, y_train, y_test= train_test_split(X, y,
        test_size=0.2, random_state=123)

In [8]: DM_train = xgb.DMatrix(data=X_train,label=y_train)

In [9]: DM_test =  xgb.DMatrix(data=X_test,label=y_test)

In [10]: params = {"booster":"gblinear","objective":"reg:linear"}

In [11]: xg_reg = xgb.train(params = params, dtrain=DM_train,
         num_boost_round=10)

In [12]: preds = xg_reg.predict(DM_test)

In [13]: rmse = np.sqrt(mean_squared_error(y_test,preds))

In [14]: print("RMSE: %f" % (rmse))
RMSE: 124326.24465
```

Regularization and base learners in XGBoost

- Regularization is a technique to discourage the complexity of the model. It does this by penalizing the loss function. This helps to solve the overfitting problem.
- Some other ways of preventing overfitting are:
  - Cross Validation
  - Drop out

Regularization and base learners in XGBoost

- Regularization is a control on model complexity
- Want models that are both accurate and as simple as possible
- Regularization parameters in XGBoost:
  - gamma - minimum loss reduction allowed for a split to occur
  - alpha - l1 regularization on leaf weights, larger values mean more regularization
  - lambda - l2 regularization on leaf weights

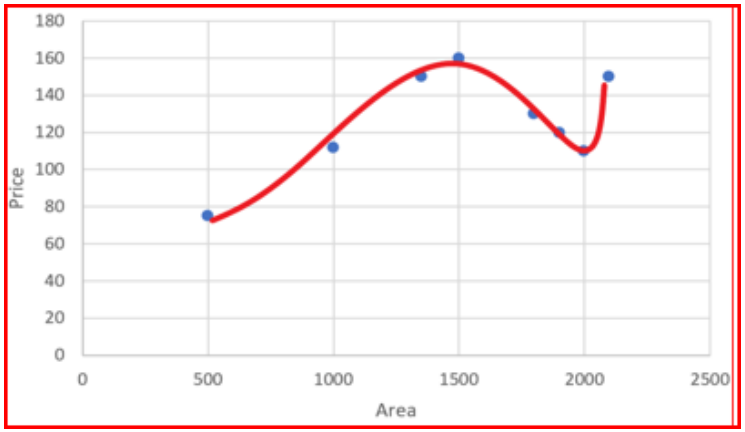
Regularization and base learners in XGBoost

- Loss function is the sum of squared difference between the actual value and the predicted value

$$L(x, y) = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

$$f(x_i) = h_\theta x = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3 + \theta_4 x_4^4$$

- As the degree of the input features increases the model becomes complex and tries to fit all the data points as shown below



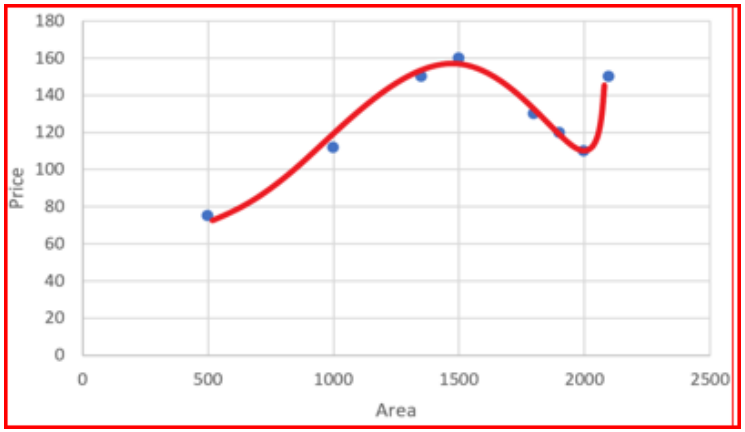
Regularization and base learners in XGBoost

- Loss function is the sum of squared difference between the actual value and the predicted value

$$L(x, y) = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

$$f(x_i) = h_\theta x = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3 + \theta_4 x_4^4$$

- As the degree of the input features increases the model becomes complex and tries to fit all the data points as shown below



Regularization and base learners in XGBoost

- When we penalize the weights θ_3 and θ_4 and make them too small, very close to zero. It makes those terms negligible and helps simplify the model.

$$f(x_i) = h_\theta x = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3 + \theta_4 x_4^4$$

$$f(x_i) = h_\theta x = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2$$

- Regularization works on assumption that smaller weights generate simpler model and thus helps avoid overfitting.

Regularization and base learners in XGBoost

- What if the input variables have an impact on the output?
- To ensure we take into account the input variables, we penalize all the weights by making them small. This also makes the model simpler and less prone to overfitting

$$L(x, y) = \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2$$

where $h_\theta x_i = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3 + \theta_4 x_4^4$

$$L(x, y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

Regularization and base learners in XGBoost

- We have added the **regularization term** to the sum of squared differences between the actual value and predicted value. Regularization term keeps the weights small making the model simpler and avoiding overfitting.
- **λ** is the penalty term or regularization parameter which determines how much to penalizes the weights.
- When **λ** is zero then the regularization term becomes zero. We are back to the original Loss function.

$$L(x, y) = \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + 0$$

Regularization and base learners in XGBoost

- When λ is large, we penalize the weights and they become close to zero. This results is a very simple model having a high bias or underfitting.

$$h_\theta x = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3 + \theta_4 x_4^4$$

Regularization and base learners in XGBoost

- ***so what is the right value for λ ?***
- It is somewhere in between 0 and a large value. we need to find an optimal value of λ so that the generalization error is small.
- A simple approach would be try different values of λ on a subsample of data, understand variability of the loss function and then use it on the entire dataset.

L1 Regularization or Lasso or L1 norm

$$L(x,y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} |\theta_i|$$

- In L1 norm we shrink the parameters to zero. When input features have weights closer to zero that leads to sparse L1 norm. In Sparse solution majority of the input features have zero weights and very few features have non zero weights.
- To predict ACT score not all input features have the same influence on the prediction. GPA score has a higher influence on ACT score than BMI of the student. L1 norm will assign a zero weight to BMI of the student as it does not have a significant impact on prediction. GPA score will have a non zero weight as it is very useful in predicting the ACT score.
- L1 regularization does feature selection. It does this by assigning insignificant input features with zero weight and useful features with a non zero weight.

L1 Regularization or Lasso or L1 norm

- In L1 regularization we penalize the absolute value of the weights. L1 regularization term is highlighted in the red box.
- Lasso produces a model that is simple, interpretable and contains a subset of input features

L2 Regularization or Ridge Regularization

$$L(x, y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

- In L2 regularization, regularization term is the sum of square of all feature weights as shown above in the equation.
- L2 regularization forces the weights to be small but does not make them zero and does non sparse solution.
- L2 is not robust to outliers as square terms blows up the error differences of the outliers and the regularization term tries to fix it by penalizing the weights
- Ridge regression performs better when all the input features influence the output and all with weights are of roughly equal size

Difference between L1 and L2 regularization

- **L1 Regularization**
  - L1 penalizes sum of absolute value of weights.
  - L1 has a sparse solution
  - L1 has multiple solutions
  - L1 has built in feature selection
  - L1 is robust to outliers
  - L1 generates model that are simple and interpretable but cannot learn complex patterns
- **L2 Regularization**
  - L2 regularization penalizes sum of square weights.
  - L2 has a non sparse solution
  - L2 has one solution
  - L2 has no feature selection
  - L2 is not robust to outliers
  - L2 gives better prediction when output variable is a function of all input features
  - L2 regularization is able to learn complex data patterns

We see that both L1 and L2 regularization have their own strengths and weakness.

https://medium.com/datadriveninvestor/l1-l2-regularization-7f1b4fe948f2

L1 Regularization in XGBoost example

```
In [1]: import xgboost as xgb
In [2]: import pandas as pd
In [3]: boston_data = pd.read_csv("boston_data.csv")
In [4]: X,y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]

In [5]: boston_dmatrix = xgb.DMatrix(data=X,label=y)
In [6]: params={"objective":"reg:linear","max_depth":4}

In [7]: l1_params = [1,10,100]
In [8]: rmses_l1=[]

In [9]: 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])

In [10]: print("Best rmse as a function of l1:")
In [11]: 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
```

Base Learners in XGBoost

- Linear Base Learner:
  - Sum of linear terms
  - Boosted model is weighted sum of linear models (thus is itself linear)
  - Rarely used
- Tree Base Learner:
  - Decision tree
  - Boosted model is weighted sum of decision trees (nonlinear)
  - Almost exclusively used in XGBoost

Fine-tuning your XGBoost model

Untuned Model Example

```
In [1]: import pandas as pd
In [2]: import xgboost as xgb
In [3]: import numpy as np
In [4]: housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
In [5]: X,y = housing_data[housing_data.columns.tolist()[:-1]],
        housing_data[housing_data.columns.tolist()[-1]]
In [6]: housing_dmatrix = xgb.DMatrix(data=X,label=y)

In [7]: untuned_params={"objective":"reg:linear"}

In [8]: untuned_cv_results_rmse = xgb.cv(dtrain=housing_dmatrix,
        params=untuned_params,nfold=4,
        metrics="rmse",as_pandas=True,seed=123)

In [9]: print("Untuned rmse: %f" %((untuned_cv_results_rmse["test-rmse-mean"]).t
Untuned rmse: 34624.229980
```

Tuned Model Example

```
In [1]: import pandas as pd
In [2]: import xgboost as xgb
In [3]: import numpy as np
In [4]: housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
In [5]: X,y = housing_data[housing_data.columns.tolist()[:-1]],
   ...: housing_data[housing_data.columns.tolist()[-1]]
In [6]: housing_dmatrix = xgb.DMatrix(data=X,label=y)

In [7]: tuned_params = {"objective":"reg:linear",'colsample_bytree': 0.3,
   ...: 'learning_rate': 0.1, 'max_depth': 5}

In [8]: 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)

In [9]: print("Tuned rmse: %f" %((tuned_cv_results_rmse["test-rmse-mean"]) \
   ...: .tail(1)))
Tuned rmse: 29812.683594
```

Tunable parameters in XGBoost

- Common tree tunable parameters
  - learning rate: learning rate/eta (https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10)
  - gamma: min loss reduction to create new tree split
  - lambda: L2 reg on leaf weights
  - alpha: L1 reg on leaf weights
  - max_depth: max depth per tree
  - subsample: % samples used per tree
  - colsample_bytree: % features used per tree

Linear tunable parameters

- lambda: L2 reg on weights
- alpha: L1 reg on weights
- lambda_bias: L2 reg term on bias
- You can also tune the number of estimators used for both base model types!

Review of Grid Search and Random Search

- Grid Search: Review
  - Search exhaustively over a given set of hyperparameters, once per set of hyperparameters
  - Number of models = number of distinct values per hyperparameter multiplied across each hyperparameter
  - Pick final model hyperparameter values that give best cross-validated evaluation metric value

Grid Search: Example

```
In [1]: import pandas as pd
In [2]: import xgboost as xgb
In [3]: import numpy as np
In [4]: from sklearn.model_selection import GridSearchCV

In [5]: housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
In [6]: X, y = housing_data[housing_data.columns.tolist()[:-1]],
   ...: housing_data[housing_data.columns.tolist()[-1]
In [7]: housing_dmatrix = xgb.DMatrix(data=X,label=y)

In [8]: gbm_param_grid = {
   ...: 'learning_rate': [0.01,0.1,0.5,0.9],
   ...: 'n_estimators': [200],
   ...: 'subsample': [0.3, 0.5, 0.9]}

In [9]: gbm = xgb.XGBRegressor()
In [10]: grid_mse = GridSearchCV(estimator=gbm,
    ...: param_grid=gbm_param_grid,
    ...: scoring='neg_mean_squared_error', cv=4, verbose=1)
In [11]: grid_mse.fit(X, y)

In [12]: print("Best parameters found: ",grid_mse.best_params_)
Best parameters found: {'learning_rate': 0.1,
'n_estimators': 200, 'subsample': 0.5}
In [13]: print("Lowest RMSE found: ", np.sqrt(np.abs(grid_mse.best_score_)))
Lowest RMSE found:  28530.1829341
```

Review of Grid Search and Random Search

- Random Search: Review
  - Create a (possibly infinite) range of hyperparameter values per hyperparameter that you would like to search over
  - Set the number of iterations you would like for the random search to continue
  - During each iteration, randomly draw a value in the range of specified values for each hyperparameter searched over and train/evaluate a model with those hyperparameters
  - After you've reached the maximum number of iterations, select the hyperparameter configuration with the best evaluated score

Random Search: Example

```
In [1]: import pandas as pd
In [2]: import xgboost as xgb
In [3]: import numpy as np
In [4]: from sklearn.model_selection import RandomizedSearchCV
In [5]: housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
In [6]: X,y = housing_data[housing_data.columns.tolist()[:-1]],
   ...: housing_data[housing_data.columns.tolist()[-1]]
In [7]: housing_dmatrix = xgb.DMatrix(data=X,label=y)

In [8]: gbm_param_grid = {
   ...: 'learning_rate': np.arange(0.05,1.05,.05),
   ...: 'n_estimators': [200],
   ...: 'subsample': np.arange(0.05,1.05,.05)}

In [9]: gbm = xgb.XGBRegressor()
In [10]: randomized_mse = RandomizedSearchCV(estimator=gbm,
    ...: param_distributions=gbm_param_grid, n_iter=25,
    ...: scoring='neg_mean_squared_error', cv=4, verbose=1)
In [11]: randomized_mse.fit(X, y)

In [12]: print("Best parameters found: ",randomized_mse.best_params_)
Best parameters found: {'subsample': 0.60000000000000009,
'n_estimators': 200, 'learning_rate': 0.20000000000000001}
In [13]: print("Lowest RMSE found: ",
    ...: np.sqrt(np.abs(randomized_mse.best_score_)))
Lowest RMSE found: 28300.2374291
```

Limits of Grid Search and Random Search

Grid Search and Random Search Limitations

Grid Search

- Number of models you must build with every additional new parameter grows very quickly

Random Search

- Parameter space to explore can be massive
- Randomly jumping throughout the space looking for a "best" result becomes a waiting game

Review of pipelines using sklearn

- Takes a list of named 2-tuples (name, pipeline_step) as input
- Tuples can contain any arbitrary scikit-learn compatible estimator or transformer object
- Pipeline implements fit/predict methods
- Can be used as input estimator into grid/randomized search and cross_val_score methods

Scikit-learn pipeline example

```
In [1]: 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

In [2]: names = ["crime","zone","industry","charles",
   ...: "no","rooms","age", "distance",
   ...: "radial","tax","pupil","aam","lower","med_price"]

In [3]: data = pd.read_csv("boston_housing.csv",names=names)

In [4]: X, y = data.iloc[:,:-1], data.iloc[:,-1]

In [5]: rf_pipeline = Pipeline[("st_scaler",
   ...: StandardScaler()),
   ...: ("rf_model",RandomForestRegressor())]

In [6]: scores = cross_val_score(rf_pipeline,X,y,
   ...: scoring="neg_mean_squared_error",cv=10)

In [7]: final_avg_rmse = np.mean(np.sqrt(np.abs(scores)))

In [8]: print("Final RMSE:", final_avg_rmse)
Final RMSE: 4.54530686529
```

Preprocessing I: LabelEncoder and OneHotEncoder

- LabelEncoder: Converts a categorical column of strings into integers
- OneHotEncoder: Takes the column of integers and encodes them as dummy variables
- Cannot be done within a pipeline

Preprocessing II: DictVectorizer

- Traditionally used in text processing
- Converts lists of feature mappings into vectors
- Need to convert DataFrame into a list of dictionary entries
- Explore the scikit-learn documentation

Incorporating xgboost into pipelines

```
In [1]: 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

In [2]: names = ["crime","zone","industry","charles","no",
   ...: "rooms","age", "distance","radial","tax",
   ...: "pupil","aam","lower","med_price"]
In [3]: data = pd.read_csv("boston_housing.csv",names=names)
In [4]: X, y = data.iloc[:,:-1], data.iloc[:,-1]

In [5]: xgb_pipeline = Pipeline[("st_scaler",
   ...: StandardScaler()),
   ...: ("xgb_model",xgb.XGBRegressor())]
In [6]: scores = cross_val_score(xgb_pipeline, X, y,
   ...:scoring="neg_mean_squared_error",cv=10)

In [7]: final_avg_rmse = np.mean(np.sqrt(np.abs(scores)))
In [8]: print("Final XGB RMSE:", final_avg_rmse)
Final RMSE: 4.02719593323
```

Additional Components Introduced For Pipelines

- sklearn_pandas:
  - DataFrameMapper - Interoperability between pandas and scikit-learn
  - CategoricalImputer - Allow for imputation of categorical variables before conversion to integers
- sklearn.preprocessing:
  - Imputer - Native imputation of numerical columns in scikit-learn
- sklearn.pipeline:
  - FeatureUnion - combine multiple pipelines of features into a single pipeline of features

Tuning xgboost hyperparameters in a pipeline

```
In [1]: 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

In [2]: names = ["crime","zone","industry","charles","no",
   ...: "rooms","age", "distance","radial","tax",
   ...: "pupil","aam","lower","med_price"]
In [3]: data = pd.read_csv("boston_housing.csv",names=names)
In [4]: X, y = data.iloc[:,:-1],data.iloc[:,-1]
In [5]: xgb_pipeline = Pipeline[("st_scaler",
   ...: StandardScaler()), ("xgb_model",xgb.XGBRegressor())]

In [6]: 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) }

In [7]: randomized_neg_mse = RandomizedSearchCV(estimator=xgb_pipeline,
   ...: param_distributions=gbm_param_grid, n_iter=10,
   ...: scoring='neg_mean_squared_error', cv=4)

In [8]: randomized_neg_mse.fit(X, y)
```

Tuning xgboost hyperparameters in a pipeline

```
In [9]: print("Best rmse: ",
   ...: np.sqrt(np.abs(randomized_neg_mse.best_score_)))
Best rmse: 3.9966784203040677

In [10]: print("Best model: ",
    ...: randomized_neg_mse.best_estimator_)
Best model:  Pipeline(steps=[('st_scaler', StandardScaler(copy=True,
with_mean=True, with_std=True)),
('xgb_model', XGBRegressor(base_score=0.5, colsample_bylevel=1,
       colsample_bytree=0.95000000000000029, gamma=0, learning_rate=0.1,
       max_delta_step=0, max_depth=8, min_child_weight=1, missing=None,
       n_estimators=100, nthread=-1, objective='reg:linear', reg_alpha=0,
       reg_lambda=1, scale_pos_weight=1, seed=0, silent=True,
       subsample=0.90000000000000013))])
```

The end.

Thanks!