

Gradient Boosting Machines (GBMs) with XGBoost

This tutorial is a part of [Machine Learning with Python: Zero to GBMs](#) and [Zero to Data Science Bootcamp by Jovian](#)



The following topics are covered in this tutorial:

- Downloading a real-world dataset from a Kaggle competition
- Performing feature engineering and prepare the dataset for training
- Training and interpreting a gradient boosting model using XGBoost
- Training with KFold cross validation and ensembling results
- Configuring the gradient boosting model and tuning hyperparameters

Let's begin by installing the required libraries.

```
!pip install numpy pandas matplotlib seaborn --quiet
```

```
!pip install jovian opendatasets xgboost graphviz lightgbm scikit-learn xgboost ligh
```

Problem Statement

This tutorial takes a practical and coding-focused approach. We'll learn gradient boosting by applying it to a real-world dataset from the [Rossmann Store Sales](#) competition on Kaggle:

Rossmann operates over 3,000 drug stores in 7 European countries. Currently, Rossmann store managers are tasked with predicting their daily sales for up to six weeks in advance. Store sales are influenced by many factors, including promotions, competition, school and state holidays, seasonality, and locality.

With thousands of individual managers predicting sales based on their unique circumstances, the accuracy of results can be quite varied. You are provided with historical sales data for 1,115 Rossmann stores. The task is to forecast the "Sales" column for the test set. Note that some stores in the dataset were temporarily closed for refurbishment.

View and download the data here: <https://www.kaggle.com/c/rossmann-store-sales/data>

▼ Downloading the Data

We can download the dataset from Kaggle directly within the Jupyter notebook using the `opendatasets` library. Make sure to [accept the competition rules](#) before executing the following cell.

```
import os
import opendatasets as od
import pandas as pd
pd.set_option("display.max_columns", 120)
pd.set_option("display.max_rows", 120)

od.download('https://www.kaggle.com/c/rossmann-store-sales')

→ Skipping, found downloaded files in "./rossmann-store-sales" (use force=True to
```

You'll be asked to provide your Kaggle credentials to download the data. Follow these instructions: <http://bit.ly/kaggle-creds>

```
os.listdir('rossmann-store-sales')

→ ['test.csv', 'train.csv', 'store.csv', 'sample_submission.csv']
```

Let's load the data into Pandas dataframes.

```
ross_df = pd.read_csv('./rossmann-store-sales/train.csv', low_memory=False)
store_df = pd.read_csv('./rossmann-store-sales/store.csv')
test_df = pd.read_csv('./rossmann-store-sales/test.csv')
submission_df = pd.read_csv('./rossmann-store-sales/sample_submission.csv')

ross_df
```

	Store	DayofWeek	Date	Sales	Customers	Open	Promo	StateHoliday	Schoo
0	1	5	2015-07-31	5263	555	1	1		0
1	2	5	2015-07-31	6064	625	1	1		0
2	3	5	2015-07-31	8314	821	1	1		0
3	4	5	2015-07-31	13995	1498	1	1		0
4	5	5	2015-07-31	4822	559	1	1		0
...
1017204	1111	2	2013-01-01	0	0	0	0	a	
1017205	1112	2	2013-01-01	0	0	0	0	a	
				...					

test_df

	Id	Store	DayofWeek	Date	Open	Promo	StateHoliday	SchoolHoliday
0	1	1	4	2015-09-17	1.0	1	0	0
1	2	3	4	2015-09-17	1.0	1	0	0
2	3	7	4	2015-09-17	1.0	1	0	0
3	4	8	4	2015-09-17	1.0	1	0	0
4	5	9	4	2015-09-17	1.0	1	0	0
...
41083	41084	1111	6	2015-08-01	1.0	0	0	0
41084	41085	1112	6	2015-08-01	1.0	0	0	0
				...				

submission_df

	Id	Sales
0	1	0
1	2	0
2	3	0
3	4	0
4	5	0
...
41083	41084	0
41084	41085	0
41085	41086	0
41086	41087	0
41087	41088	0

41088 rows × 2 columns

store_df

	Store	StoreType	Assortment	CompetitionDistance	CompetitionOpenSinceMonth
0	1	c	a	1270.0	9.0
1	2	a	a	570.0	11.0
2	3	a	a	14130.0	12.0
3	4	c	c	620.0	9.0
4	5	a	a	29910.0	4.0
...
1110	1111	a	a	1900.0	6.0
1111	1112	c	c	1880.0	4.0
1112	1113	a	c	9260.0	NaN
1113	1114	a	c	870.0	NaN
1114	1115	d	c	5350.0	NaN

1115 rows × 10 columns

EXERCISE: Read the data description provided on the competition page to understand what the values in each column of `store_df` represent.

Start coding or generate with AI.

Let's merge the information from `store_df` into `train_df` and `test_df`.

```
merged_df = ross_df.merge(store_df, how='left', on='Store')
merged_test_df = test_df.merge(store_df, how='left', on='Store')
```

merged_df

	Store	DayOfWeek	Date	Sales	Customers	Open	Promo	StateHoliday	Schoo
0	1	5	2015-07-31	5263	555	1	1		0
1	2	5	2015-07-31	6064	625	1	1		0
2	3	5	2015-07-31	8314	821	1	1		0
3	4	5	2015-07-31	13995	1498	1	1		0
4	5	5	2015-07-31	4822	559	1	1		0
...
1017204	1111	2	2013-01-01	0	0	0	0	a	
1017205	1112	2	2013-01-01	0	0	0	0	a	
1017206	1113	2	2013-01-01	0	0	0	0	a	
1017207	1114	2	2013-01-01	0	0	0	0	a	
1017208	1115	2	2013-01-01	0	0	0	0	a	

1017209 rows × 18 columns

EXERCISE: Perform exploratory data analysis and visualization on the dataset. Study the distribution of values in each column, and their relationship with the target column Sales .

Start coding or generate with AI.

Start coding or generate with AI.

Let's save our work before continuing.

```
jovian.commit()
```

→ [jovian] Committed successfully! <https://jovian.ai/aakashns/python-gradient-boos>
['https://jovian.ai/aakashns/python-gradient-boosting-machines'](https://jovian.ai/aakashns/python-gradient-boosting-machines)

▼ Preprocessing and Feature Engineering

Let's take a look at the available columns, and figure out if we can create new columns or apply any useful transformations.

```
merged_df.info()
```

→ <class 'pandas.core.frame.DataFrame'>
Int64Index: 1017209 entries, 0 to 1017208
Data columns (total 18 columns):
 # Column Non-Null Count Dtype
 -- -- -- --
 0 Store 1017209 non-null int64
 1 DayOfWeek 1017209 non-null int64
 2 Date 1017209 non-null object
 3 Sales 1017209 non-null int64
 4 Customers 1017209 non-null int64
 5 Open 1017209 non-null int64
 6 Promo 1017209 non-null int64
 7 StateHoliday 1017209 non-null object
 8 SchoolHoliday 1017209 non-null int64
 9 StoreType 1017209 non-null object
 10 Assortment 1017209 non-null object
 11 CompetitionDistance 1014567 non-null float64
 12 CompetitionOpenSinceMonth 693861 non-null float64
 13 CompetitionOpenSinceYear 693861 non-null float64
 14 Promo2 1017209 non-null int64
 15 Promo2SinceWeek 509178 non-null float64
 16 Promo2SinceYear 509178 non-null float64
 17 PromoInterval 509178 non-null object
 dtypes: float64(5), int64(8), object(5)

memory usage: 147.5+ MB

▼ Date

First, let's convert Date to a datecolumn and extract different parts of the date.

```
def split_date(df):
    df['Date'] = pd.to_datetime(df['Date'])
    df['Year'] = df.Date.dt.year
    df['Month'] = df.Date.dt.month
    df['Day'] = df.Date.dt.day
    df['WeekOfYear'] = df.Date.dt.isocalendar().week
```

```
split_date(merged_df)
split_date(merged_test_df)
```

```
merged_df
```

	Store	DayOfWeek	Date	Sales	Customers	Open	Promo	StateHoliday	Schoo
0	1	5	2015-07-31	5263	555	1	1		0
1	2	5	2015-07-31	6064	625	1	1		0
2	3	5	2015-07-31	8314	821	1	1		0
3	4	5	2015-07-31	13995	1498	1	1		0
4	5	5	2015-07-31	4822	559	1	1		0
...
1017204	1111	2	2013-01-01	0	0	0	0	a	
1017205	1112	2	2013-01-01	0	0	0	0	a	
1017206	1113	2	2013-01-01	0	0	0	0	a	
1017207	1114	2	2013-01-01	0	0	0	0	a	
1017208	1115	2	2013-01-01	0	0	0	0	a	

1017209 rows × 22 columns

▼ Store Open/Closed

Next, notice that the sales are zero whenever the store is closed.

```
merged_df[merged_df.Open == 0].Sales.value_counts()
```

0	172817
Name:	Sales, dtype: int64

Instead of trying to model this relationship, it would be better to hard-code it in our predictions, and remove the rows where the store is closed. We won't remove any rows from the test set, since we need to make predictions for every row.

```
merged_df = merged_df[merged_df.Open == 1].copy()
```

▼ Competition

Next, we can use the columns `CompetitionOpenSince [Month/Year]` columns from `store_df` to compute the number of months for which a competitor has been open near the store.

```
def comp_months(df):
    df['CompetitionOpen'] = 12 * (df.Year - df.CompetitionOpenSinceYear) + (df.Month
    df['CompetitionOpen'] = df['CompetitionOpen'].map(lambda x: 0 if x < 0 else x).f
    comp_months(merged_df)
    comp_months(merged_test_df)

merged_df
```

	Store	DayOfWeek	Date	Sales	Customers	Open	Promo	StateHoliday	Schoo
0	1	5	2015-07-31	5263	555	1	1		0
1	2	5	2015-07-31	6064	625	1	1		0
2	3	5	2015-07-31	8314	821	1	1		0
3	4	5	2015-07-31	13995	1498	1	1		0
4	5	5	2015-07-31	4822	559	1	1		0
...
1016776	682	2	2013-01-01	3375	566	1	0	a	
1016827	733	2	2013-01-01	10765	2377	1	0	a	
1016863	769	2	2013-01-01	5035	1248	1	0	a	
1017042	948	2	2013-01-01	4491	1039	1	0	a	
1017190	1097	2	2013-01-01	5961	1405	1	0	a	

844392 rows × 23 columns

Let's view the results of the new columns we've created.

```
merged_df[['Date', 'CompetitionDistance', 'CompetitionOpenSinceYear', 'CompetitionOp
```

	Date	CompetitionDistance	CompetitionOpenSinceYear	CompetitionOpenSinceMonth
21092	2015-07-13	3740.0	2002.0	
521375	2014-03-21	190.0	2011.0	
962441	2013-02-19	21930.0		NaN
572789	2014-02-03	10890.0		2005.0
744556	2013-09-02	7160.0		2012.0
375144	2014-08-05	3390.0		NaN
339654	2014-09-12	1850.0		2014.0
79686	2015-05-21	7160.0		2012.0
853087	2013-05-28	18540.0		NaN
91114	2015-05-11	2020.0		2014.0
798663	2013-07-16	3490.0		2011.0
651052	2013-11-25	3250.0		NaN
222496	2015-01-13	2490.0		2012.0

▼ Additional Promotion

We can also add some additional columns to indicate how long a store has been running `Promo2` and whether a new round of `Promo2` starts in the current month.

```
def check_promo_month(row):
    month2str = {1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May', 6:'Jun',
                 7:'Jul', 8:'Aug', 9:'Sept', 10:'Oct', 11:'Nov', 12:'Dec'}
    try:
        months = (row['PromoInterval'] or '').split(',')
        if row['Promo2Open'] and month2str[row['Month']] in months:
            return 1
    except:
        pass
```

```
else:  
    return 0  
except Exception:  
    return 0  
  
def promo_cols(df):  
    # Months since Promo2 was open  
    df['Promo2Open'] = 12 * (df.Year - df.Promo2SinceYear) + (df.WeekOfYear - df.Promo2Week)  
    df['Promo2Open'] = df['Promo2Open'].map(lambda x: 0 if x < 0 else x).fillna(0) *  
    # Whether a new round of promotions was started in the current month  
    df['IsPromo2Month'] = df.apply(check_promo_month, axis=1) * df['Promo2']  
  
promo_cols(merged_df)  
promo_cols(merged_test_df)
```

Let's view the results of the columns we've created.

```
merged_df[['Date', 'Promo2', 'Promo2SinceYear', 'Promo2SinceWeek', 'PromoInterval',
```

	Date	Promo2	Promo2SinceYear	Promo2SinceWeek	PromoInterval	Promo2Open
271634	2014-11-24	1	2011.0	18.0	Feb,May,Aug,Nov	42.885246
66318	2015-06-02	1	2009.0	45.0	Jan,Apr,Jul,Oct	66.950820
63049	2015-06-05	0	NaN	NaN	NaN	0.000000
588334	2014-01-20	1	2011.0	14.0	Jan,Apr,Jul,Oct	33.704918
500994	2014-04-08	1	2014.0	10.0	Mar,Jun,Sept,Dec	1.147541
79333	2015-05-21	1	2014.0	18.0	Feb,May,Aug,Nov	12.688525
396138	2014-07-14	1	2013.0	36.0	Jan,Apr,Jul,Oct	10.393443
540035	2014-03-04	1	2011.0	14.0	Jan,Apr,Jul,Oct	35.081967
723881	2013-09-21	1	2012.0	40.0	Jan,Apr,Jul,Oct	11.540984
435210	2014-06-06	0	NaN	NaN	NaN	0.000000
346549	2014-09-05	0	NaN	NaN	NaN	0.000000
150873	2015-03-18	0	NaN	NaN	NaN	0.000000
835446	2013-06-13	0	NaN	NaN	NaN	0.000000

The features related to competition and promotion are now much more useful.

▼ Input and Target Columns

Let's select the columns that we'll use for training.

`merged_df.columns`

→ Index(['Store', 'DayOfWeek', 'Date', 'Sales', 'Customers', 'Open', 'Promo', 'StateHoliday', 'SchoolHoliday', 'StoreType', 'Assortment', 'CompetitionDistance', 'CompetitionOpenSinceMonth', 'CompetitionOpenSinceYear', 'Promo2', 'Promo2SinceWeek', 'Promo2SinceYear', 'PromoInterval', 'Year', 'Month', 'Day'],

```
'WeekOfYear', 'CompetitionOpen', 'Promo2Open', 'IsPromo2Month'],
dtype='object')

input_cols = ['Store', 'DayOfWeek', 'Promo', 'StateHoliday', 'SchoolHoliday',
    'StoreType', 'Assortment', 'CompetitionDistance', 'CompetitionOpen',
    'Day', 'Month', 'Year', 'WeekOfYear', 'Promo2',
    'Promo2Open', 'IsPromo2Month']
target_col = 'Sales'

inputs = merged_df[input_cols].copy()
targets = merged_df[target_col].copy()

test_inputs = merged_test_df[input_cols].copy()
```

Let's also identify numeric and categorical columns. Note that we can treat binary categorical columns (0/1) as numeric columns.

```
numeric_cols = ['Store', 'Promo', 'SchoolHoliday',
    'CompetitionDistance', 'CompetitionOpen', 'Promo2', 'Promo2Open', 'IsP
    'Day', 'Month', 'Year', 'WeekOfYear', ]
categorical_cols = ['DayOfWeek', 'StateHoliday', 'StoreType', 'Assortment']
```

▼ Impute missing numerical data

```
inputs[numeric_cols].isna().sum()
```

```
→ Store      0
  Promo      0
  SchoolHoliday  0
  CompetitionDistance  0
  CompetitionOpen  0
  Promo2      0
  Promo2Open   0
  IsPromo2Month  0
  Day         0
  Month       0
  Year        0
  WeekOfYear   0
  dtype: int64
```

```
test_inputs[numeric_cols].isna().sum()
```

```
→ Store      0
  Promo      0
  SchoolHoliday  0
  CompetitionDistance  0
```

```
CompetitionOpen      0
Promo2              0
Promo2Open          0
IsPromo2Month       0
Day                 0
Month               0
Year                0
WeekOfYear          0
dtype: int64
```

Seems like competition distance is the only missing value, and we can simply fill it with the highest value (to indicate that competition is very far away).

```
max_distance = inputs.CompetitionDistance.max()
```

```
inputs['CompetitionDistance'].fillna(max_distance, inplace=True)
test_inputs['CompetitionDistance'].fillna(max_distance, inplace=True)
```

▼ Scale Numeric Values

Let's scale numeric values to the 0 to 1 range.

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler().fit(inputs[numeric_cols])

inputs[numeric_cols] = scaler.transform(inputs[numeric_cols])
test_inputs[numeric_cols] = scaler.transform(test_inputs[numeric_cols])
```

▼ Encode Categorical Columns

Index	Categorical column
1	Cat A
2	Cat B
3	Cat C

Index	Cat A	Cat B	Cat C
1	1	0	0
2	0	1	0
3	0	0	1

Let's one-hot encode categorical columns.

```
from sklearn.preprocessing import OneHotEncoder
```

```
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore').fit(inputs[categorical_cols])
encoded_cols = list(encoder.get_feature_names(categorical_cols))

inputs[encoded_cols] = encoder.transform(inputs[categorical_cols])
test_inputs[encoded_cols] = encoder.transform(test_inputs[categorical_cols])
```

Finally, let's extract out all the numeric data for training.

```
X = inputs[numeric_cols + encoded_cols]
X_test = test_inputs[numeric_cols + encoded_cols]
```

We haven't created a validation set yet, because we'll use K-fold cross validation.

EXERCISE: Look through the notebooks created by participants in the Kaggle competition and apply some other ideas for feature engineering.

[https://www.kaggle.com/c/rossmann-store-sales/code?
competitionId=4594&sortBy=voteCount](https://www.kaggle.com/c/rossmann-store-sales/code?competitionId=4594&sortBy=voteCount)

Start coding or generate with AI.

Start coding or generate with AI.

Let's save our work before continuing.

```
jovian.commit()
```

→ [jovian] Updating notebook "akashns/python-gradient-boosting-machines" on <https://jovian.ai/akashns/python-gradient-boos>
[jovian] Committed successfully! <https://jovian.ai/akashns/python-gradient-boos> '<https://jovian.ai/akashns/python-gradient-boosting-machines>'

Gradient Boosting

We're now ready to train our gradient boosting machine (GBM) model. Here's how a GBM model works:

1. The average value of the target column and uses as an initial prediction every input.
2. The residuals (difference) of the predictions with the targets are computed.
3. A decision tree of limited depth is trained to **predict just the residuals** for each input.

4. Predictions from the decision tree are scaled using a parameter called the learning rate (this prevents overfitting)
5. Scaled predictions from the tree are added to the previous predictions to obtain the new and improved predictions.
6. Steps 2 to 5 are repeated to create new decision trees, each of which is trained to predict just the residuals from the previous prediction.

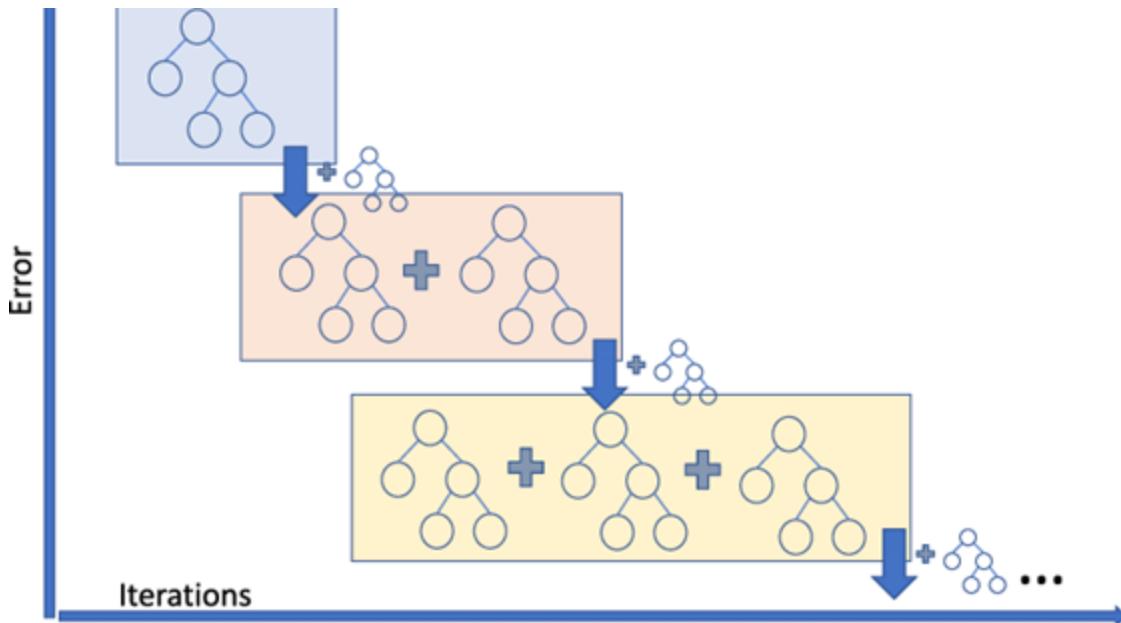
The term "gradient" refers to the fact that each decision tree is trained with the purpose of reducing the loss from the previous iteration (similar to gradient descent). The term "boosting" refers to the general technique of training new models to improve the results of an existing model.

EXERCISE: Can you describe in your own words how a gradient boosting machine is different from a random forest?

For a mathematical explanation of gradient boosting, check out the following resources:

- [XGBoost Documentation](#)
- [Video Tutorials on StatQuest](#)

Here's a visual representation of gradient boosting:



▼ Training

To train a GBM, we can use the `XGBRegressor` class from the [XGBoost](#) library.

```
from xgboost import XGBRegressor
```

```
?XGBRegressor
```

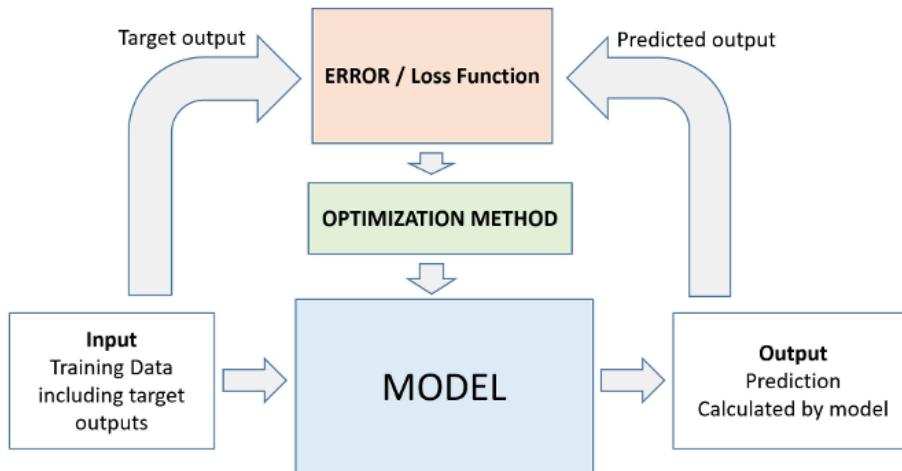
```
model = XGBRegressor(random_state=42, n_jobs=-1, n_estimators=20, max_depth=4)
```

Let's train the model using `model.fit`.

```
%%time
model.fit(X, targets)
```

```
→ CPU times: user 44.7 s, sys: 1.53 s, total: 46.2 s
Wall time: 3.23 s
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.300000012, max_delta_step=0, max_depth=4,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=20, n_jobs=-1, num_parallel_tree=1, random_state=42,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

EXERCISE: Explain how the `.fit` method of `XGBRegressor` applies the iterative machine learning workflow to train the model using the training data.



▼ Prediction

We can now make predictions and evaluate the model using `model.predict`.

```
preds = model.predict(X)
```

```
preds
```

```
→ array([ 8127.9404,  7606.919 ,  8525.857 , ...,  6412.8247,  9460.068 ,
          10302.145 ], dtype=float32)
```

✓ Evaluation

Let's evaluate the predictions using RMSE error.

```
from sklearn.metrics import mean_squared_error

def rmse(a, b):
    return mean_squared_error(a, b, squared=False)

rmse(preds, targets)
→ 2377.752008804669
```

Start coding or generate with AI.

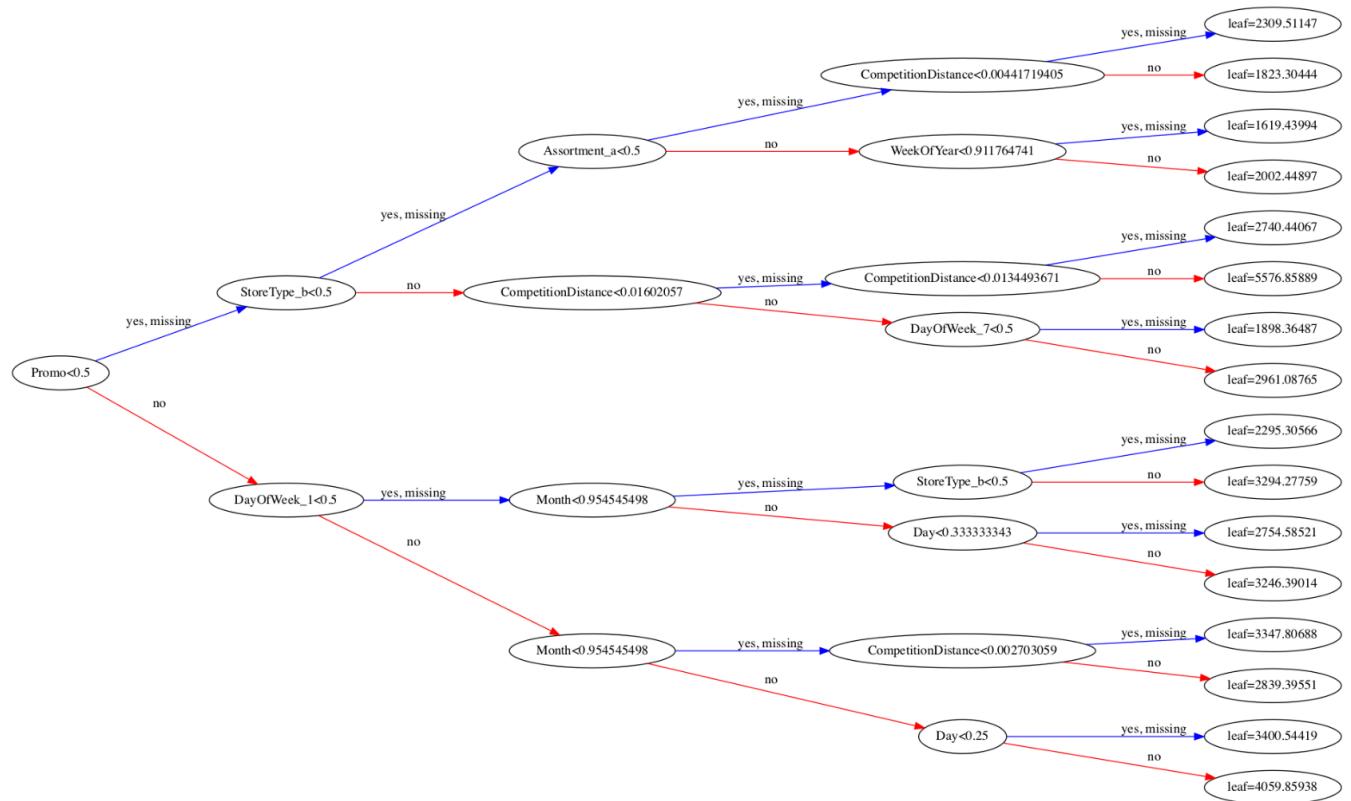
✓ Visualization

We can visualize individual trees using `plot_tree` (note: this requires the `graphviz` library to be installed).

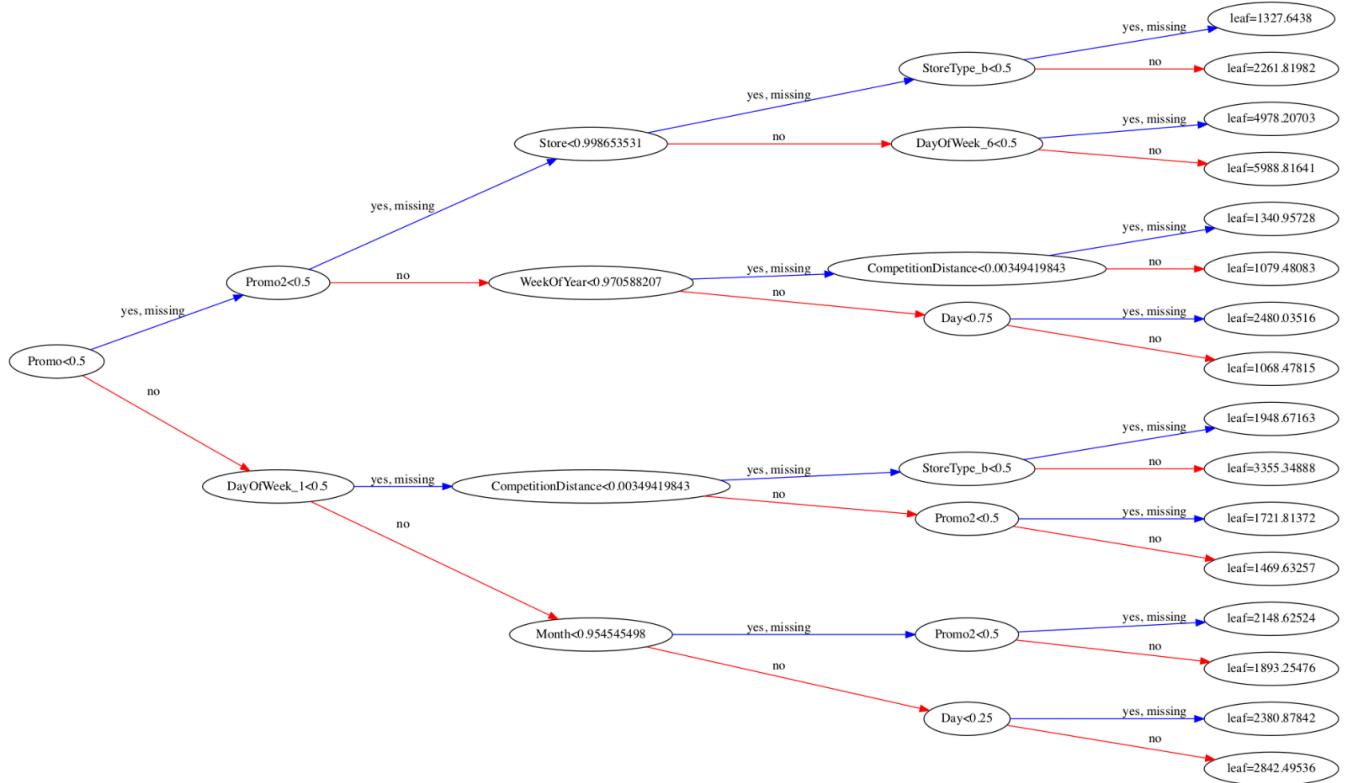
```
import matplotlib.pyplot as plt
from xgboost import plot_tree
from matplotlib.pylab import rcParams
%matplotlib inline

rcParams['figure.figsize'] = 30,30

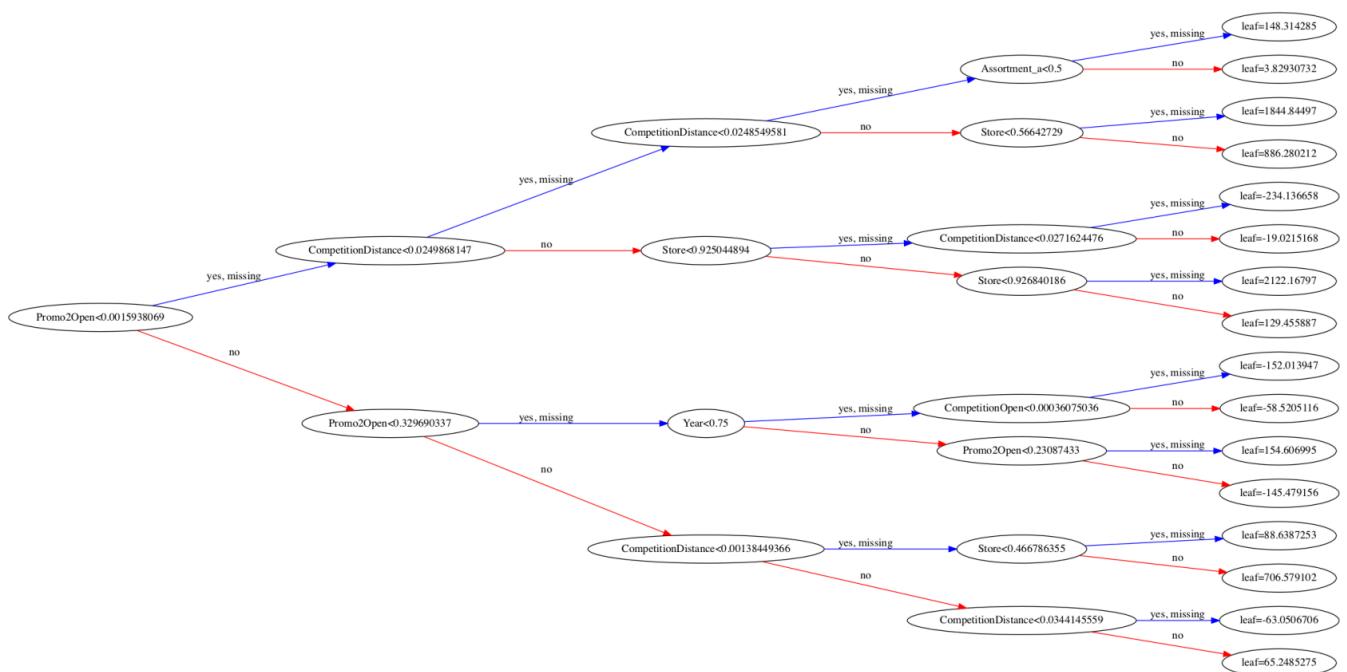
plot_tree(model, rankdir='LR');
```



```
plot_tree(model, rankdir='LR', num_trees=1);
```



```
plot_tree(model, rankdir='LR', num_trees=19);
```



Notice how the trees only compute residuals, and not the actual target value. We can also visualize the tree as text.

```
trees = model.get_booster().get_dump()
```

`len(trees)`

20

```
print(trees[0])
```

```
→ 0: [Promo<0.5] yes=1,no=2,missing=1
    1: [StoreType_b<0.5] yes=3,no=4,missing=3
        3: [Assortment_a<0.5] yes=7,no=8,missing=7
            7: [CompetitionDistance<0.00441719405] yes=15,no=16,missing=15
                15:leaf=2309.51147
                16:leaf=1823.30444
            8: [WeekOfYear<0.911764741] yes=17,no=18,missing=17
                17:leaf=1619.43994
                18:leaf=2002.44897
        4: [CompetitionDistance<0.01602057] yes=9,no=10,missing=9
            9: [CompetitionDistance<0.0134493671] yes=19,no=20,missing=19
                19:leaf=2740.44067
                20:leaf=5576.85889
        10: [DayOfWeek_7<0.5] yes=21,no=22,missing=21
            21:leaf=1898.36487
            22:leaf=2961.08765
    2: [DayOfWeek_1<0.5] yes=5,no=6,missing=5
        5: [Month<0.954545498] yes=11,no=12,missing=11
            11: [StoreType_b<0.5] yes=23,no=24,missing=23
                23:leaf=2295.30566
                24:leaf=3294.27759
        12: [Day<0.333333343] yes=25,no=26,missing=25
            25:leaf=2754.58521
            26:leaf=3246.39014
    6: [Month<0.954545498] yes=13,no=14,missing=13
        13: [CompetitionDistance<0.002703059] yes=27,no=28,missing=27
            27:leaf=3347.80688
            28:leaf=2839.39551
        14: [Day<0.25] yes=29,no=30,missing=29
            29:leaf=3400.54419
            30:leaf=4059.85938
```

Feature importance

Just like decision trees and random forests, XGBoost also provides a feature importance score for each column in the input.

```
importance_df = pd.DataFrame({
    'feature': X.columns,
    'importance': model.feature_importances_
}).sort_values('importance', ascending=False)

importance_df.head(10)
```

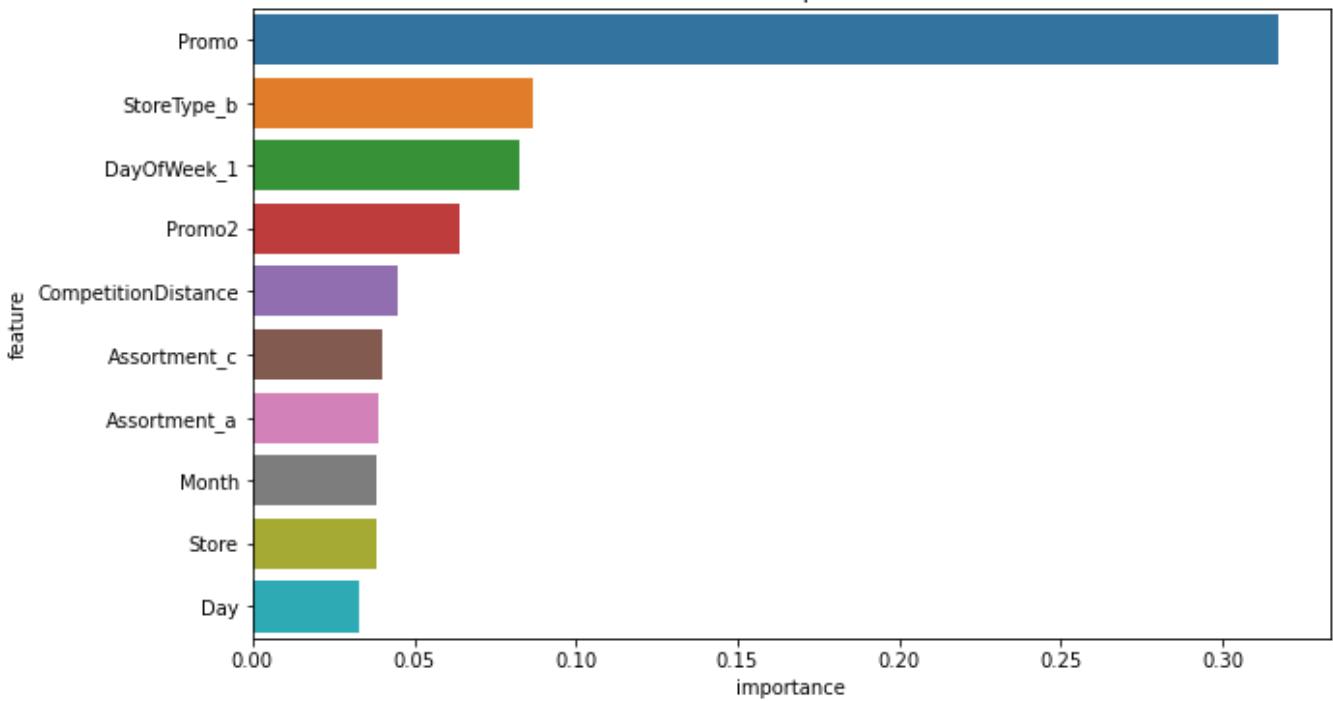
**feature importance**

1	Promo	0.317473
24	StoreType_b	0.086472
12	DayOfWeek_1	0.082269
5	Promo2	0.063986
3	CompetitionDistance	0.045053
29	Assortment_c	0.040226
27	Assortment_a	0.038759
9	Month	0.038493
0	Store	0.038119
8	Day	0.033209

```
import seaborn as sns
plt.figure(figsize=(10,6))
plt.title('Feature Importance')
sns.barplot(data=importance_df.head(10), x='importance', y='feature');
```



Feature Importance



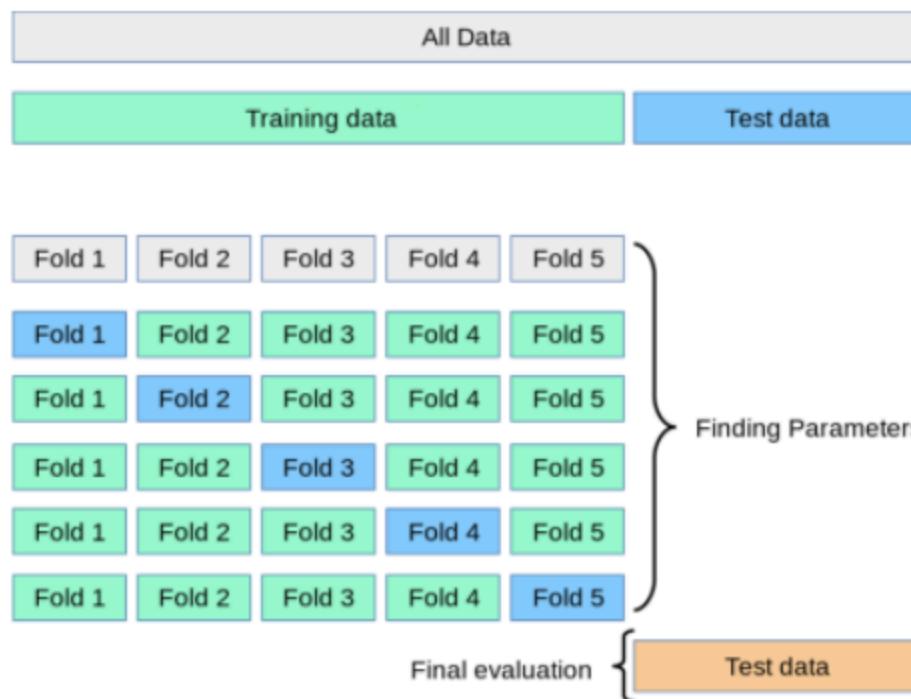
Let's save our work before continuing.

```
jovian.commit()
```

```
→ [jovian] Updating notebook "akashns/python-gradient-boosting-machines" on https://jovian.ai/akashns/python-gradient-boos
[jovian] Committed successfully! https://jovian.ai/akashns/python-gradient-boos'
```

▼ K Fold Cross Validation

Notice that we didn't create a validation set before training our XGBoost model. We'll use a different validation strategy this time, called K-fold cross validation ([source](#)):



Scikit-learn provides utilities for performing K fold cross validation.

```
from sklearn.model_selection import KFold
```

Let's define a helper function `train_and_evaluate` which trains a model the given parameters and returns the trained model, training error and validation error.

```
def train_and_evaluate(X_train, train_targets, X_val, val_targets, **params):
    model = XGBRegressor(random_state=42, n_jobs=-1, **params)
    model.fit(X_train, train_targets)
    train_rmse = rmse(model.predict(X_train), train_targets)
    val_rmse = rmse(model.predict(X_val), val_targets)
    return model, train_rmse, val_rmse
```

Now, we can use the `KFold` utility to create the different training/validations splits and train a separate model for each fold.

```
kfold = KFold(n_splits=5)

models = []

for train_idxs, val_idxs in kfold.split(X):
    X_train, train_targets = X.iloc[train_idxs], targets.iloc[train_idxs]
    X_val, val_targets = X.iloc[val_idxs], targets.iloc[val_idxs]
    model, train_rmse, val_rmse = train_and_evaluate(X_train,
                                                    train_targets,
                                                    X_val,
                                                    val_targets,
                                                    max_depth=4,
                                                    n_estimators=20)
    models.append(model)
print('Train RMSE: {}, Validation RMSE: {}'.format(train_rmse, val_rmse))

→ Train RMSE: 2352.216448531526, Validation RMSE: 2424.6228916973314
Train RMSE: 2406.709513789309, Validation RMSE: 2451.9646038059277
Train RMSE: 2365.7354745443067, Validation RMSE: 2336.984157073758
Train RMSE: 2366.4732092777763, Validation RMSE: 2460.8995475901697
Train RMSE: 2379.3752997474626, Validation RMSE: 2440.665320626728
```

Let's also define a function to average predictions from the 5 different models.

```
import numpy as np

def predict_avg(models, inputs):
    return np.mean([model.predict(inputs) for model in models], axis=0)

preds = predict_avg(models, X)

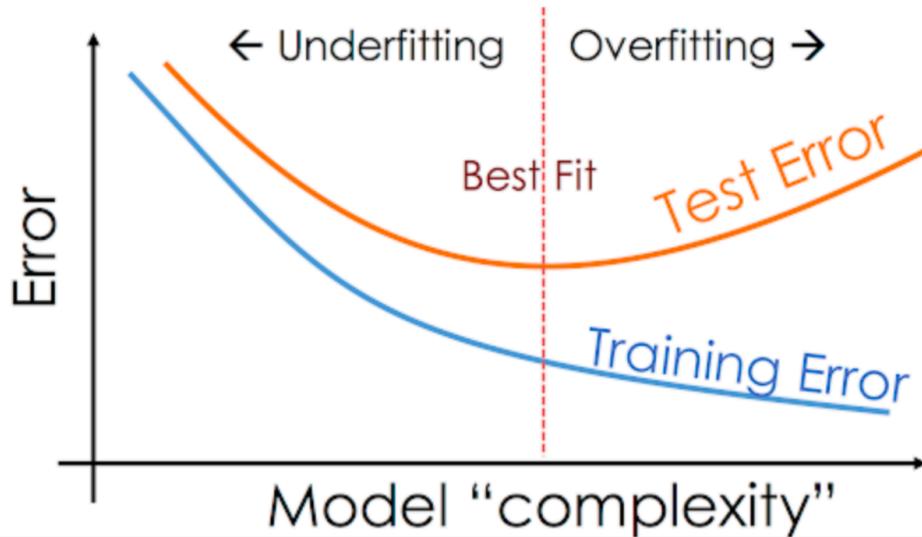
preds

→ array([8021.374 , 7577.715 , 8747.863 , ..., 7615.0303, 7924.784 ,
       9600.297 ], dtype=float32)
```

We can now use `predict_avg` to make predictions for the test set.

▼ Hyperparameter Tuning and Regularization

Just like other machine learning models, there are several hyperparameters we can adjust to adjust the capacity of model and reduce overfitting.



Check out the following resources to learn more about hyperparameter supported by XGBoost:

- https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.XGBRegressor
- <https://xgboost.readthedocs.io/en/latest/parameter.html>

model

?XGBRegressor

Here's a helper function to test hyperparameters with K-fold cross validation.

```
def test_params_kfold(n_splits, **params):
    train_rmses, val_rmses, models = [], [], []
    kfold = KFold(n_splits)
    for train_idxs, val_idxs in kfold.split(X):
        X_train, train_targets = X.iloc[train_idxs], targets.iloc[train_idxs]
        X_val, val_targets = X.iloc[val_idxs], targets.iloc[val_idxs]
        model, train_rmse, val_rmse = train_and_evaluate(X_train, train_targets, X_v
        models.append(model)
        train_rmses.append(train_rmse)
        val_rmses.append(val_rmse)
    print('Train RMSE: {}, Validation RMSE: {}'.format(np.mean(train_rmses), np.mean
    return models
```

Since it may take a long time to perform 5-fold cross validation for each set of parameters we wish to try, we'll just pick a random 10% sample of the dataset as the validation set.

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_val, train_targets, val_targets = train_test_split(X, targets, test_size=
```

```
def test_params(**params):
    model = XGBRegressor(n_jobs=-1, random_state=42, **params)
    model.fit(X_train, train_targets)
    train_rmse = rmse(model.predict(X_train), train_targets)
    val_rmse = rmse(model.predict(X_val), val_targets)
    print('Train RMSE: {}, Validation RMSE: {}'.format(train_rmse, val_rmse))
```

▼ n_estimators

The number of trees to be created. More trees = greater capacity of the model.

```
test_params(n_estimators=10)
```

→ Train RMSE: 2353.663414241198, Validation RMSE: 2356.229854163455

```
test_params(n_estimators=30)
```

→ Train RMSE: 1911.2460991462647, Validation RMSE: 1915.6472657649233

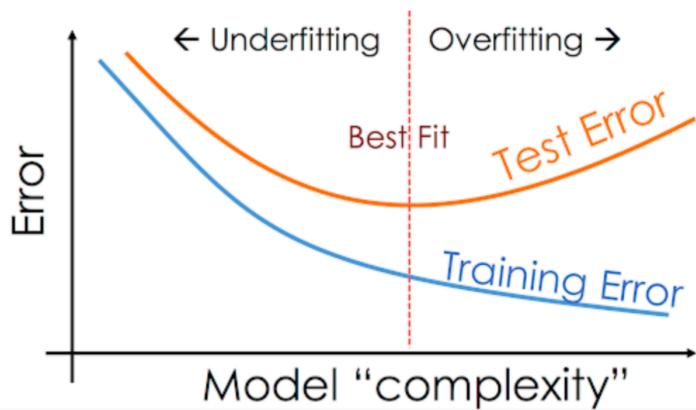
```
test_params(n_estimators=100)
```

→ Train RMSE: 1185.4562733444168, Validation RMSE: 1193.1959233994555

```
test_params(n_estimators=240)
```

→ Train RMSE: 895.8322715342299, Validation RMSE: 910.1409179651465

EXERCISE: Experiment with different values of n_estimators, plot a graph of the training and validation error and determine the best value for n_estimators .



Start coding or generate with AI.

Start coding or generate with AI.

▼ max_depth

As you increase the max depth of each tree, the capacity of the tree increases and it can capture more information about the training set.

```
test_params(max_depth=2)
```

→ Train RMSE: 2354.679896364511, Validation RMSE: 2351.1681023945134

```
test_params(max_depth=5)
```

→ Train RMSE: 1456.0644283203644, Validation RMSE: 1457.2885926259178

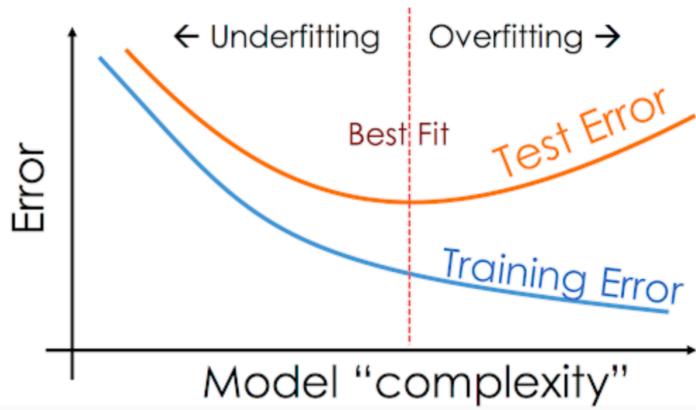
```
test_params(max_depth=10)
```

→ Train RMSE: 691.2025514846819, Validation RMSE: 777.7774318826166

Start coding or generate with AI.

Start coding or generate with AI.

EXERCISE: Experiment with different values of `max_depth`, plot a graph of the training and validation error and determine the optimal.



▼ learning_rate

The scaling factor to be applied to the prediction of each tree. A very high learning rate (close to 1) will lead to overfitting, and a low learning rate (close to 0) will lead to underfitting.

```
test_params(n_estimators=50, learning_rate=0.01)
```

→ Train RMSE: 5044.000166628424, Validation RMSE: 5039.631093589314

```
test_params(n_estimators=50, learning_rate=0.1)
```

→ Train RMSE: 2167.858504538518, Validation RMSE: 2167.2963858379153

```
test_params(n_estimators=50, learning_rate=0.3)
```

→ Train RMSE: 1559.3736718929556, Validation RMSE: 1566.68780047217

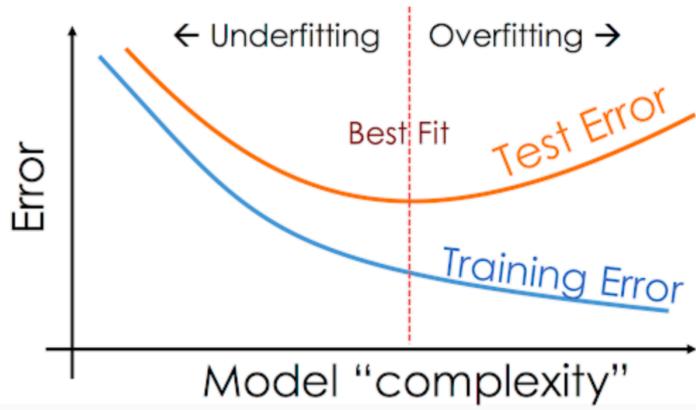
```
test_params(n_estimators=50, learning_rate=0.9)
```

→ Train RMSE: 1115.8929050462332, Validation RMSE: 1124.0787360992879

```
test_params(n_estimators=50, learning_rate=0.99)
```

→ Train RMSE: 1141.0686753930636, Validation RMSE: 1153.9657795546302

EXERCISE: Experiment with different values of `learning_rate`, plot a graph of the training and validation error and determine the optimal.



Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

▼ booster

Instead of using Decision Trees, XGBoost can also train a linear model for each iteration. This can be configured using `booster`.

```
test_params(booster='gblinear')
```

→ Train RMSE: 217054.65872292817, Validation RMSE: 217077.36665393168

Clearly, a linear model is not well suited for this dataset.

EXERCISE: Experiment with other hyperparameters like `gamma`, `min_child_weight`, `max_delta_step`, `subsample`, `colsample_bytree` etc. and find their optimal values. Learn more about them here:

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

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

EXERCISE: Train a model with your best hyperparameters and evaluate its performance using 5-fold cross validation.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Let's save our work before continuing.

```
jovian.commit()
```

→

Putting it Together and Making Predictions

Let's train a final model on the entire training set with custom hyperparameters.

```
model = XGBRegressor(n_jobs=-1, random_state=42, n_estimators=1000,  
                     learning_rate=0.2, max_depth=10, subsample=0.9,
```



```
%%time
model.fit(X, targets)

CPU times: user 1h 25min 32s, sys: 1min 35s, total: 1h 27min 8s
Wall time: 5min 54s
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=0.7, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.2, max_delta_step=0, max_depth=10,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=1000, n_jobs=-1, num_parallel_tree=1,
              random_state=42,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=0.9,
              tree_method='exact', validate_parameters=1, verbosity=None)
```

Now that the model is trained, we can make predictions on the test set.

```
test_preds = model.predict(X_test)
```

Let's add the predictions into `submission_df`.

```
submission_df['Sales'] = test_preds
```

Recall, however, if the store is not open, then the sales must be 0. Thus, wherever the value of `Open` in the test set is 0, we can set the sales to 0. Also, there are some missing values for `Open` in