Introduction

Until now, you have learned the original boosting algorithm, AdaBoost.

In this session, we will learn another popular boosting algorithm - **Gradient Boosting** and a modification of Gradient Boosting called **XGBoost** which is widely used in the industry. We will also briefly discuss some of the more advanced algorithms recently introduced.

Towards the end, we will go through the implementation of the different algorithms on a Kaggle Dataset in Python.

In this session

Let's look at the broad flow of this session.

Prerequisites

Understanding how gradient descent works is a prerequisite for this session. You would have gone through the topic in the Linear Regression module.

Guidelines for in-module questions

The in-video and in-content questions for this module are not graded.

People you will hear from in this session

Subject Matter Expert:

Prof G Srinivasaraghavan

Professor, IIIT- B

The International Institute of Information Technology, Bangalore, commonly known as IIIT Bangalore, is a premier national graduate school in India. Founded in 1999, it offers Integrated M.Tech., M.Tech., M.S. (Research) and PhD programs in the field of Information Technology.

Anjali Rajvanshi

Sr. Subject Matter Expert, upGrad

Snehansu Sekhar Sahu

Sr. Research Engineer in ML & AI at American Express

Snehanshu has more than 6 years of experience and has worked with companies like Qualcomm Inc, Infoedge solutions & American Express. He is currently part of the ML & AI Research Group @ AmEx.

Report an error

NEXT

Understanding Gradient Boosting - I

Understanding Gradient Boosting - I

Gradient Boosting like AdaBoost trains many models in a gradual, additive, and sequential manner. But the major difference between the two is how they identify & handle the shortcomings of weak learners. In AdaBoost, we provide more say or weight to those data points which are misclassified/ wrongly predicted earlier. Gradient boosting performs the same by using gradients in the loss function.

In the next video, Anjali will explain the fundamentals of the Gradient boosting machine and how does it work.

The loss function is a measure indicating how good the model is able to fit the underlying data. This varies from one problem statement to another problem statement and thus would depend on what we are trying to optimize. In the case of a regression problem, the loss function would be the error between true and predicted target values. For classification problems, the loss function would be a measure of how good our predictive model is at classifying each sample point that we have in our dataset.

Let's now understand GBM with the help of a numerical example for a regression problem.

To summarize here are the broader points on how does a GBM learn:

- Build the first weak learner using a sample from the training data; we will consider a decision tree as the weak learner or the base model. It may not necessarily be a stump, can grow a bigger tree but will still be weak i.e. still not be fully grown.
- Then the predictions are made on the training data using the decision tree just built.
- The gradient, in our case the residuals are computed and these **residuals are the new response or target values for the next weak learner.**
- A new weak learner is built with the residuals as the target values and a sample of observations from the original training data.
- Add the predictions obtained from the current weak learner to the predictions obtained from all the previous weak learners. The predictions obtained at each step are multiplied by the learning rate so that no single model makes a huge contribution to the ensemble thereby avoiding overfitting. Essentially, with the addition of each weak learner, the model takes a very small step in the right direction.
- The next weak learner fits on the residuals obtained till now and these steps are repeated, either for a prespecified number of weak learners or if the model starts overfitting i.e. it starts to capture the niche patterns of the training data.
- GBM makes the final prediction by simply adding up the predictions from all the weak learners (multiplied by the learning rate).

Report an error

Understanding Gradient Boosting - I

Gradient Boosting like AdaBoost trains many models in a gradual, additive, and sequential manner. But the major difference between the two is how they identify & handle the shortcomings of weak learners. In AdaBoost, we provide more say or weight to those data points which are misclassified/ wrongly predicted earlier. Gradient boosting performs the same by using gradients in the loss function.

In the next video, Anjali will explain the fundamentals of the Gradient boosting machine and how does it work.

The loss function is a measure indicating how good the model is able to fit the underlying data. This varies from one problem statement to another problem statement and thus would depend on what we are trying to optimize. In the case of a regression problem, the loss function would be the error between true and predicted target values. For classification problems, the loss function would be a measure of how good our predictive model is at classifying each sample point that we have in our dataset.

Let's now understand GBM with the help of a numerical example for a regression problem.

To summarize here are the broader points on how does a GBM learn:

- Build the first weak learner using a sample from the training data; we will consider a decision tree as the weak learner or the base model. It may not necessarily be a stump, can grow a bigger tree but will still be weak i.e. still not be fully grown.
- Then the predictions are made on the training data using the decision tree just built.
- The gradient, in our case the residuals are computed and these **residuals are the new response or target values for the next weak learner.**
- A new weak learner is built with the residuals as the target values and a sample of observations from the original training data.
- Add the predictions obtained from the current weak learner to the predictions obtained from all the previous weak learners. The predictions obtained at each step are multiplied by the learning rate so that no single model makes a huge contribution to the ensemble thereby avoiding overfitting. Essentially, with the addition of each weak learner, the model takes a very small step in the right direction.
- The next weak learner fits on the residuals obtained till now and these steps are repeated, either for a prespecified number of weak learners or if the model starts overfitting i.e. it starts to capture the niche patterns of the training data.
- GBM makes the final prediction by simply adding up the predictions from all the weak learners (multiplied by the learning rate).

Report an error

Understanding Gradient Boosting - II

In this session, we will go through the algorithm of gradient boosting in a regression setting. With the intuition learned from the previous videos, let us see how this translates to the algorithm behind the model.

At any iteration t, we repeat the following steps in the Gradient Boosting scheme of things:

1. Initialize a crude initial function Fo as $argmin \sum_{t=i}^{T} \boldsymbol{L}(\boldsymbol{y_i}, \hat{\boldsymbol{y}})$

- This will be the average value which will form our first prediciton.

1. For m = 1 to M (where M is the number of trees)
    1. Calculate the pseudo-residuals $r_{im} = -\frac{(\partial \boldsymbol{L}(\boldsymbol{y_i}, \boldsymbol{F}(\boldsymbol{x_i}))}{\partial \boldsymbol{F}(\boldsymbol{x_i})}$, where $F(x_i) = F_{m-1}(x_i)$, the pseudo residuals are the negative gradients for all data points
    2. Fit a base learner $h_m(x)$ to the pseudo-residuals, i.e train it using the training set $\sum_{i=1}^{n} (x_i, r_{im})$. Here the pseudo residuals are used as the response variable.
    3. Compute the step magnitude multiplier γ_m (in case of tree models, compute a γ_m different for every leaf/prediction). In other words, introducing step multiplier results in a small step in the right direction.
    $\gamma_m = argmin \sum_{i=1}^{n} L(y_i, (F_{m-1} + \gamma * h_m(x_i))$
    4. Compute the next model $F_m = F_{m-1}(x_i) + \gamma_m * h_m(x_i)$
2. The final model is $F_M(x)$

We see here that the gradient boosting algorithm essentially starts off with a crude model that gives a certain output y_{avg}, which is the mean of all the target values.
The next function F_1 is developed such that it models the **residuals**, $y_i - y_{avg}$.

The point to note here is that the new model F_1, builds a regression model on the feature vectors of their **residual**. Hence the target variable for F_1 is the residual, $y_i - y_i^0$ and not the actual target variable y_i. We can write the residual $y_i - y_i^0$ as $y_i - F_0(x_i)$.

Question 1/3 Mandatory

Gradient Boosting Steps

In continuation to the above video, after we fit the model F_1 on the residuals $y_i - F_0(x_i)$, the prediction we get for x_i is $F_1(x_i)$. What do you think is the residual now?

- $y_i - F_0(x_i) - F_1(x_i)$ ✓ Correct

  Feedback:

  The target variable for F_1 was $y_i - F_0(x_i)$. Hence, the residual will be target variable - predicted value which is $y_i - F_0(x_i) - F_1(x_i)$.

- $y_i - F_1(x_i)$

Your answer is Correct. Attempt 1 of 1 Continue

Now that you have got a brief understanding of how gradient boosting works, let's study what the *gradient* in gradient boosting is.

Report an error

Question 2/3 Mandatory

Gradient Boosting Steps

Now, like we created a new function F_1 and trained it on $y_i - F_0(x_i)$, we create a new function F_2 as the next model to be trained in the Gradient Boosting Algorithm. What do you think F_2 will train on?

- y_i
- $y_i - F_1(x_i)$
- $y_i - F_0(x_i) - F_1(x_i)$ ✓ Correct

Feedback:
In every step/iteration of Gradient Boosting, we fit the new model on the residual of the corresponding step. So here F_2 will fit on the residual $y_i - F_0(x_i) - F_1(x_i)$.

Your answer is Correct. Attempt 2 of 2 Continue

Now that you have got a brief understanding of how gradient boosting works, let's study what the *gradient* in gradient boosting is.

Question 3/3 Mandatory

General Expression for Residual

Let us generalize the above expression for a model F_t. What is the general expression of the target variable the model F_t trains on? Assume, y is the initial target variable.

- y - $[F_0 + F_{t-1}]$
- y - $[F_o + F_1 + F_2 + \ldots\ldots\ldots + F_{t-2} + F_{t-1}]$ ✓ Correct

Feedback:

F_1 trains on $y - F_0$, F_2 trains on $y - F_0 - F_1 = y - [F_0 + F_1]$, and so on. In general, F_t trains on the reisduals generated by the model F_{t-1} which is y - $[F_0 + F_1 + F_2 + \ldots\ldots\ldots + F_{t-2} + F_{t-1}]$.

Your answer is Correct. Attempt 1 of 1 Continue

Now that you have got a brief understanding of how gradient boosting works, let's study what the *gradient* in gradient boosting is.

Report an error

Gradient in Gradient Boosting

We got an intuition of how the gradient boosting process helps in reducing the error with each iteration. To summarise, the gradient boosting algorithm has two steps at each iteration:

1. Find the residual and fit a new model on the residual
2. Add the new model to the older model and continue the next iteration

Let's now see how this can be seen as a **gradient descent problem**.

Comprehension - Gradient Descent

You have learned that in the gradient descent algorithm, a function $G(x)$ decreases fastest around a point a if one goes along the negative gradient of $G(x)$. It follows the iterative process:

$$a_{n+1} = a_n - \lambda \frac{\mathrm{d}G(x)}{\mathrm{d}x}$$

Now if the function is $G(x, y)$, we have the following formulation:

$$x_{n+1} = x_n - \lambda \frac{\partial G(x,y)}{\partial x}$$

Keeping these in mind answer the following questions:

Question 1/2 Mandatory

Gradient Descent of univariate function

Let $g(x) = x^3 + e^{-x}$.

Which direction should we move to minimize $g(x)$ at a point with the x-coordinate x?

- $3x^2 + e^{-x}$
- $3x^2 - e^{-x}$
- $e^{-x} - 3x^2$ ✓ Correct

Feedback:
We should be moving in the direction of negative gradient which is $e^{-x} - 3x^2$

Your answer is Correct. Attempt 2 of 2 Continue

We see that the negative gradient of the square loss function gives the residue. So the strategy of closing in on the residues at each step can be seen as taking a **step towards the negative gradient of the loss function**. We can observe here that gradient descent helps in generalising the boosting process.

Please **note** that this explanation and the aforementioned process of closing in on the residues is only valid for square error loss function.

Additional Reading

Report an error

PREVIOUS
Understanding Gradient Boosting - II

NEXT
Gradient Boosting Algorithm

Keeping these in mind answer the following questions:

Question 2/2 Mandatory

Gradient Descent of bivariate function

Let, $g(x) = x^3 + y^2$.

Which direction should we move in to minimize this function at a point (x,y)?

Note that the answer will be a vector in which i represents the x-direction and j represents the y-direction.

- $x^3i + y^2j$
- -($3x^2i + 2yj$) ✓ Correct

  Feedback:
  We have -$\frac{\partial}{\partial x}g(x,y)$ = -$3x^2$ and -$\frac{\partial}{\partial y}g(x,y)$ = -$2y$. Hence, we move in the direction -($3x^2i + 2yj$)
- $xi + yj$

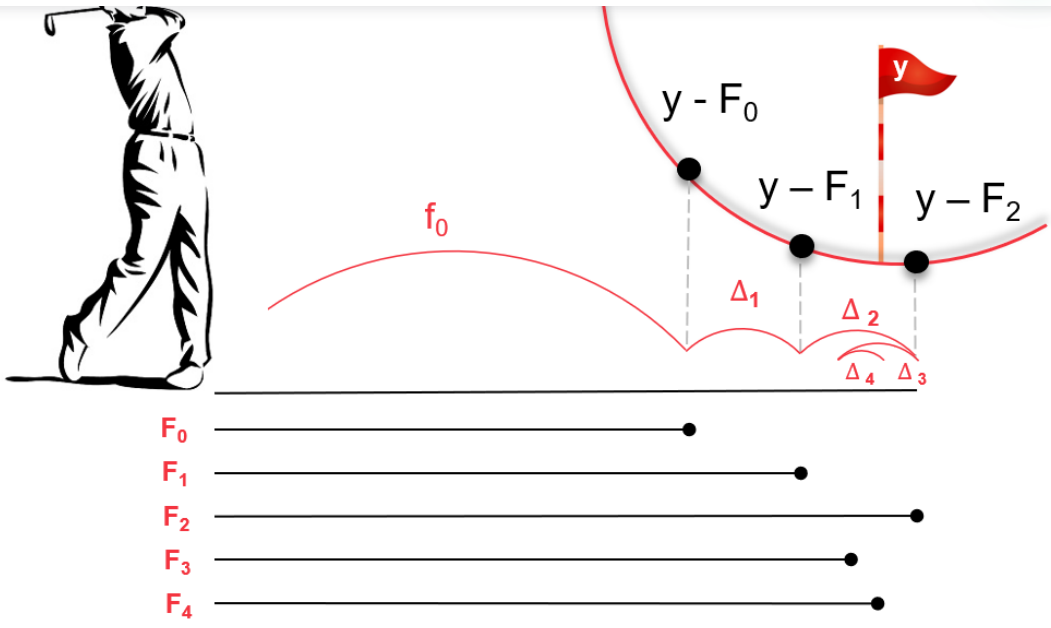
Your answer is Correct. Attempt 1 of 2 Continue

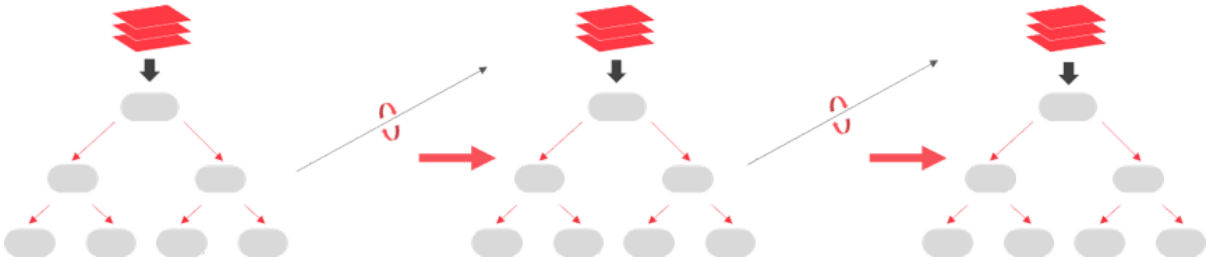
Gradient Boosting Algorithm

The **negative gradients** at an iteration t act as a **target variable** to fit a model h_m and for a regression setting with a square loss function, this target variable was the residue.

To explain this in just words, at each iteration we add an incremental model, which fits on the negative gradients of the loss function evaluated at current target values. This incremental model can be a linear regression model or a decision tree or any other model.

We stop when we see that the gradients are very close to zero. For a regression setting, this means that when the residuals are very close to zero, we stop iterating. Here, γ_m, known as the **step multiplier,** is typically between 0 and 1.
In other words, introducing step multiplier results in a small step in the right direction. γ corresponds to the hyperparameter α in terms of the utility (both determine by what amount an update should be made), but here γ is trainable while α is a hyperparameter.





Here at each iteration, a new model(weak learner) is added and with each new addition, we can observe a reduction in the pseudo residuals which is an indication that we are moving in the right direction of the target values.

Refer to the original documentation for both Regression & Classification,

You can try out GBM for classification using the notebook below

Attrition prediction using GBM Download

You can try out GBM for regression using the notebook below

Sales predicton using GBM Download

The dataset used for the Sales prediction can be downloaded here

Additional Reading

For practical implementation please go through this blog.

Visit this kernel to see how each parameter is can be tuned for model improvement.

To check how we can do gradient boosting from scratch, you can check this article.

To play with different hyperparameters in Gradient Boosting, visit this website.

Report an error

PREVIOUS
Gradient in Gradient Boosting

NEXT
Understanding XGBoost

The issue of keeping one's employees happy and satisfied is a perennial and age-old challenge. If an employee you have invested so much time and money leaves for "greener pastures", then this would mean that you would have to spend even more time and money to hire somebody else.

This project is based on a hypothetical dataset downloaded from IBM HR Analytics Employee Attrition & Performance. It has 1,470 data points (rows) and 35 features (columns) describing each employee's background and characteristics; and labelled (supervised learning) with whether they are still in the company or whether they have gone to work somewhere else.

Problem statement To understand and determine how these factors relate to workforce attrition.

Dataset: https://www.kaggle.com/patelprashant/employee-attrition (https://www.kaggle.com/patelprashant/employee-attrition)

In [2]:

```
#Importing the libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import plotly.offline as py
import plotly.graph_objs as go
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (accuracy_score, log_loss, confusion_matrix)
#Suppressing warnings
import warnings
warnings.filterwarnings('ignore')
```

In [4]:

```
#Importing  the Dataset
df = pd.read_csv('WA_Fn-UseC_-HR-Employee-Attrition.csv')
```

In [5]:

```
df.head(3)
```

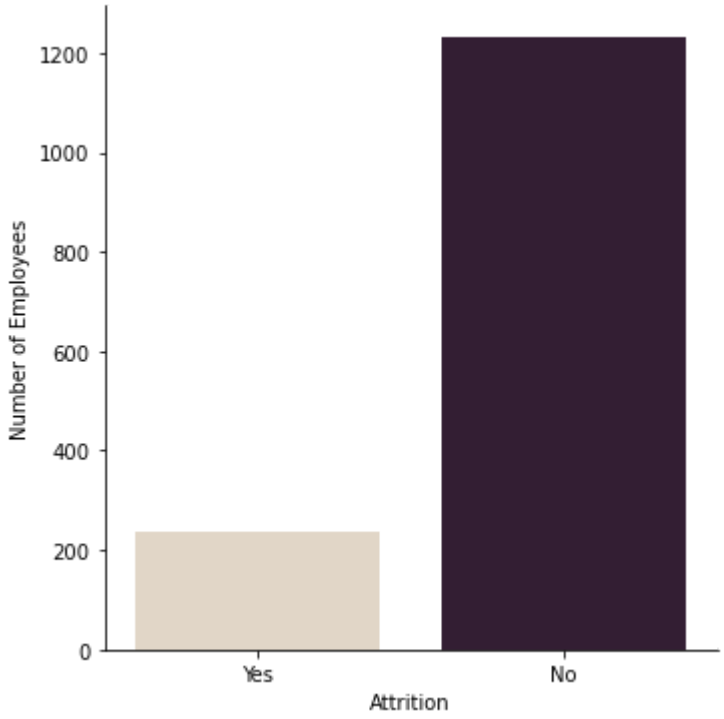
Out[5]:

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	Edu
0	41	Yes	Travel_Rarely	1102	Sales	1	2	L
1	49	No	Travel_Frequently	279	Research & Development	8	1	L
2	37	Yes	Travel_Rarely	1373	Research & Development	2	2	

3 rows × 35 columns

In [6]:

```
#Checking the number of 'Yes' and 'No' in 'Attrition'
ax = sns.catplot(x="Attrition", kind="count", palette="ch:.25", data=df);
ax.set(xlabel = 'Attrition', ylabel = 'Number of Employees')
plt.show()
```



any missing values in the dataframe.

In [7]:

```
#Identifying columns with missing information
missing_col = df.columns[df.isnull().any()].values
print('The missing columns in the dataset are: ',missing_col)
```

```
The missing columns in the dataset are:  []
```

Step 2 - Feature Engineering

The numeric and categorical fields need to be treated separately and the target field needs to be separated from the training dataset. The following few steps separate the numeric and categorical fields and drops the target field 'Attrition' from the feature set.

In [8]:

```
#Extracting the Numeric and Categorical features
df_num = pd.DataFrame(data = df.select_dtypes(include = ['int64']))
df_cat = pd.DataFrame(data = df.select_dtypes(include = ['object']))
print("Shape of Numeric: ",df_num.shape)
print("Shape of Categorical: ",df_cat.shape)
```

```
Shape of Numeric:  (1470, 26)
Shape of Categorical:  (1470, 9)
```

2.1 Encoding Categorical Fields

The categorical fields have been encoded using the get_dummies() function of Pandas.

In [9]:

```
#Dropping 'Attrition' from df_cat before encoding
df_cat = df_cat.drop(['Attrition'], axis=1)

#Encoding using Pandas' get_dummies
df_cat_encoded = pd.get_dummies(df_cat)
df_cat_encoded.head(5)
```

Out[9]:

	BusinessTravel_Non-Travel	BusinessTravel_Travel_Frequently	BusinessTravel_Travel_Rarely	Departr
0	0	0	1	
1	0	1	0	
2	0	0	1	
3	0	1	0	
4	0	0	1	

5 rows × 29 columns

2.2 Scaling Numeric Fields

The numeric fields have been scaled next for best results. StandardScaler() has been used for the same. Post scaling of the numeric features, they are merged with the categorical ones.

In [10]:

```
#Using StandardScaler to scale the numeric features
standard_scaler = StandardScaler()
df_num_scaled = standard_scaler.fit_transform(df_num)
df_num_scaled = pd.DataFrame(data = df_num_scaled, columns = df_num.columns, index = df_num
print("Shape of Numeric After Scaling: ",df_num_scaled.shape)
print("Shape of categorical after Encoding: ",df_cat_encoded.shape)
```

```
Shape of Numeric After Scaling:  (1470, 26)
Shape of categorical after Encoding:  (1470, 29)
```

In [11]:

```
#Combining the Categorical and Numeric features
df_transformed_final = pd.concat([df_num_scaled,df_cat_encoded], axis = 1)
print("Shape of final dataframe: ",df_transformed_final.shape)
```

```
Shape of final dataframe:  (1470, 55)
```

In [12]:

```
#Extracting the target variable - 'Attrition'
target = df['Attrition']

#Mapping 'Yes' to 1 and 'No' to 0
map = {'Yes':1, 'No':0}
target = target.apply(lambda x: map[x])

print("Shape of target: ",target.shape)

#Copying into commonly used fields for simplicity
X = df_transformed_final #Features
y = target #Target
```

```
Shape of target:  (1470,)
```

2.3 Train and Test Split

The data is next split into training and test dataset using the train_test_split functionality of sklearn.

In [13]:

```
#Splitting into Train and Test dataset in 90-10 ratio
X_train, X_test, y_train, y_test = train_test_split(X,y,train_size = 0.8, random_state = 0,
print("Shape of X Train: ",X_train.shape)
print("Shape of X Test: ",X_test.shape)
print("Shape of y Train: ",y_train.shape)
print("Shape of y Test: ",y_test.shape)
```

```
Shape of X Train:  (1176, 55)
Shape of X Test:  (294, 55)
Shape of y Train:  (1176,)
Shape of y Test:  (294,)
```

Step 3 - Model Fitting

In [16]:

```
#Using Gradient Boosting to predict 'Attrition' and create the Trees to identify important
gbm = GradientBoostingClassifier(n_estimators = 200, max_features = 0.7, learning_rate = 0.

#Fitting Model
gbm.fit(X_train, y_train)

#pred
y_pred = gbm.predict(X_test)
```

In [17]:

```
print('Accuracy of the model is:  ',accuracy_score(y_test, y_pred))
```

```
Accuracy of the model is:   0.8469387755102041
```

In [18]:

```
#Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print('The confusion Matrix : \n',cm)
```

```
The confusion Matrix :
 [[241   6]
 [ 39   8]]
```

Step 4 - Visualisation and Identification of Important Features

Here, I have used the 'feature_importances_' array of the Gradient Boosting Model to ascertain the most important features for the prediction of 'Attrition'.

In [20]:

```
# Scatter plot
trace = go.Scatter(
    y = gbm.feature_importances_,
    x = df_transformed_final.columns.values,
    mode='markers',
    marker=dict(
        sizemode = 'diameter',
        sizeref = 1,
        size = 12,
        color = gbm.feature_importances_,
        colorscale='Portland',
        showscale=True
    ),
    text = df_transformed_final.columns.values
)
data = [trace]

layout= go.Layout(
    autosize= True,
    title= 'Model Feature Importance',
    hovermode= 'closest',
     xaxis= dict(
         ticklen= 5,
         showgrid=False,
        zeroline=False,
        showline=False
     ),
    yaxis=dict(
        title= 'Feature Importance',
        showgrid=False,
        zeroline=False,
        ticklen= 5,
        gridwidth= 2
    ),
    showlegend= False
)
fig = go.Figure(data=data, layout=layout)
py.iplot(fig,filename='scatter')
```

Online property companies offer valuations of houses using machine learning techniques. The dataset consisted of historic data of houses sold between May 2014 to May 2015. We will predict the sales of houses in King County with an accuracy of at least 75-80% and understand which factors are responsible for higher property value - $650K and above."

Problem statement The aim of this report is to predict the house sales in King County, Washington State, USA using different models (Adaboost & GBM)

Dataset: https://www.kaggle.com/shivachandel/kc-house-data (https://www.kaggle.com/shivachandel/kc-house-data)

STEP 1: IMPORTING LIBRARIES

In [30]:

```
import numpy as np
import pandas as pd
import xgboost
import math
import seaborn as sns
import matplotlib
import matplotlib.pyplot as plt
from __future__ import division
from scipy.stats import pearsonr
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
#from sklearn.cross_validation import ShuffleSplit
from sklearn.metrics import explained_variance_score
from time import time
from sklearn.metrics import r2_score
import os
from sklearn.model_selection import train_test_split

#from sklearn import cross_validation, tree, linear_model
```

In [9]:

```
data = pd.read_csv('kc_house_data.csv')
```

In [10]:

```
# Copying data to another dataframe df_train for our convinience so that original dataframe

df_train=data.copy()
df_train.rename(columns ={'price': 'SalePrice'}, inplace =True)
```

In [11]:

```
# Now lets see the first five rows of the data
data.head()
```

Out[11]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0

5 rows × 21 columns

STEP 2: DATA CLEANING AND PREPROCESSING

In [12]:

```
print(len(data))
# Check the number of features in the data set
print(len(data.columns))
# Check the data types of each column
print(data.dtypes)
```

```
21613
21
id                int64
date             object
price           float64
bedrooms          int64
bathrooms       float64
sqft_living       int64
sqft_lot          int64
floors          float64
waterfront        int64
view              int64
condition         int64
grade             int64
sqft_above        int64
sqft_basement     int64
yr_built          int64
yr_renovated      int64
zipcode           int64
lat             float64
long            float64
sqft_living15     int64
sqft_lot15        int64
dtype: object
```

In [15]:

```
# Check any number of columns with NaN or missing values
print(data.isnull().any().sum(), ' / ', len(data.columns))
```

```
0  /  21
```

In [14]:

```
# Check any number of data points with NaN
print(data.isnull().any(axis=1).sum(), ' / ', len(data))
```

```
0  /  21613
```

STEP 3 : FINDING CORRELATION

In [16]:

```
# As id and date columns are not important to predict price so we are discarding it for fin
features = data.iloc[:,3:].columns.tolist()
target = data.iloc[:,2].name
```

In [17]:

```
# Finding Correlation of price woth other variables to see how many variables are strongly
correlations = {}
for f in features:
    data_temp = data[[f,target]]
    x1 = data_temp[f].values
    x2 = data_temp[target].values
    key = f + ' vs ' + target
    correlations[key] = pearsonr(x1,x2)[0]
```

In [18]:

```
# Printing all the correlated features value with respect to price which is target variable
data_correlations = pd.DataFrame(correlations, index=['Value']).T
data_correlations.loc[data_correlations['Value'].abs().sort_values(ascending=False).index]
```

Out[18]:

	Value
sqft_living vs price	0.702035
grade vs price	0.667434
sqft_above vs price	0.605567
sqft_living15 vs price	0.585379
bathrooms vs price	0.525138
view vs price	0.397293
sqft_basement vs price	0.323816
bedrooms vs price	0.308350
lat vs price	0.307003
waterfront vs price	0.266369
floors vs price	0.256794
yr_renovated vs price	0.126434
sqft_lot vs price	0.089661
sqft_lot15 vs price	0.082447
yr_built vs price	0.054012
zipcode vs price	-0.053203
condition vs price	0.036362
long vs price	0.021626

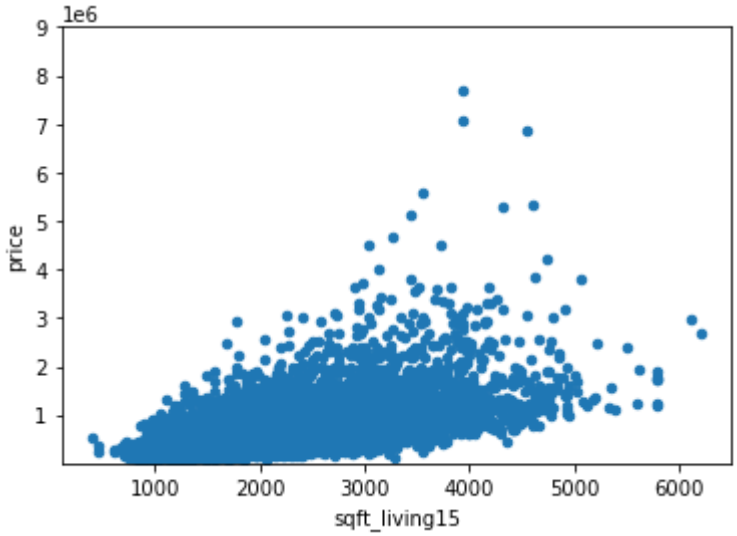
As zipcode is negatively correlated with sales price , so we can remove it for sales price prediction.

**STEP 4 : EDA or DATA VISUALIZATION **

Let's explore the data

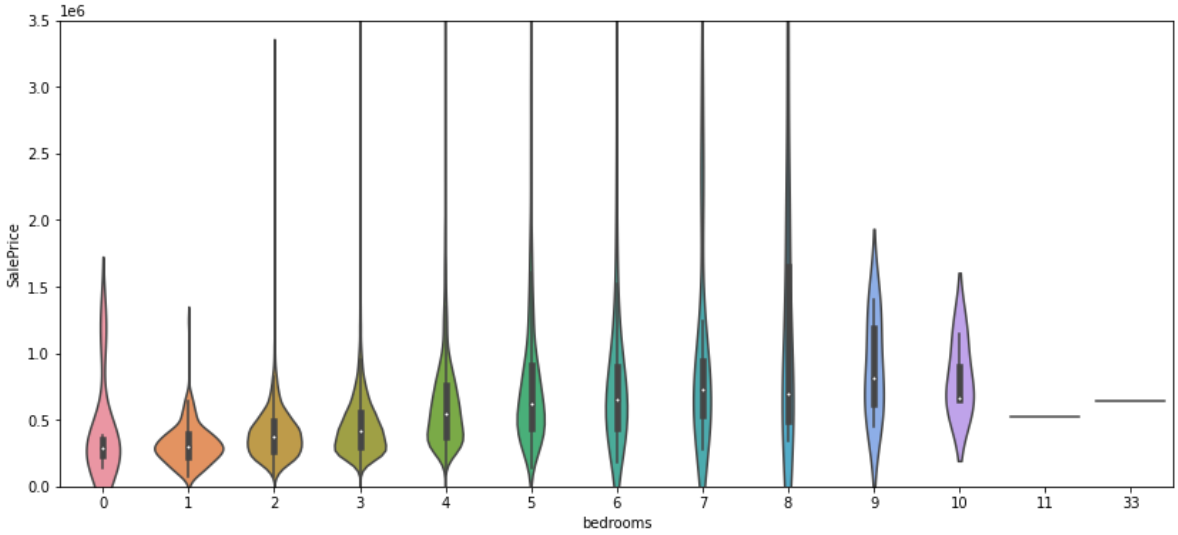
In [19]:

```
var = 'sqft_living15'
data = pd.concat([data['price'], data[var]], axis=1)
data.plot.scatter(x=var, y='price', ylim=(3,9000000));
```



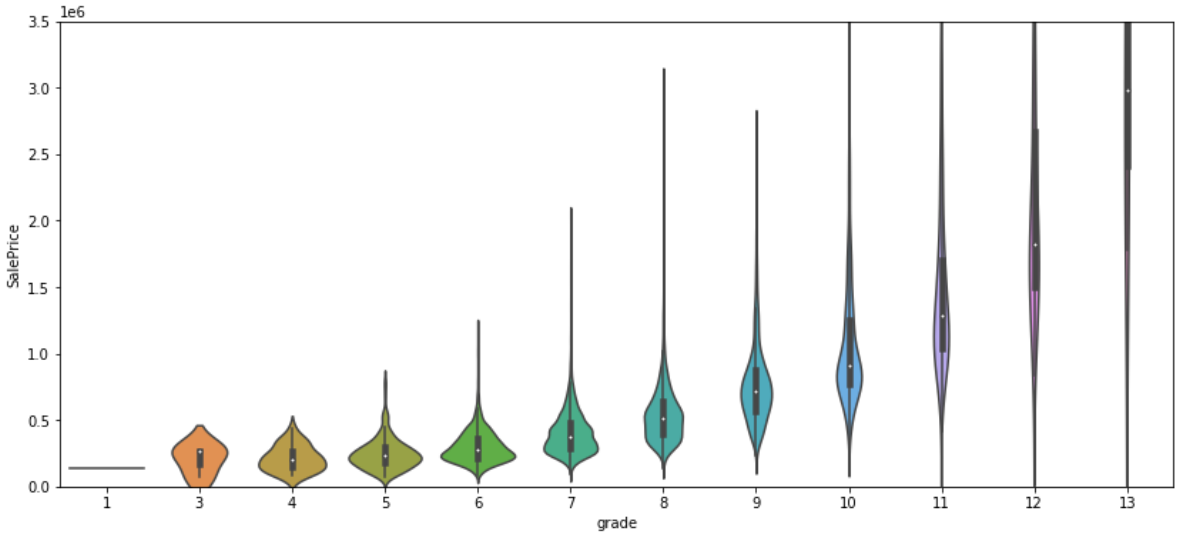
In [20]:

```
var = 'bedrooms'
data = pd.concat([df_train['SalePrice'], df_train[var]], axis=1)
f, ax = plt.subplots(figsize=(14, 6))
fig = sns.violinplot(x=var, y="SalePrice", data=data)
fig.axis(ymin=0, ymax=3500000);
```



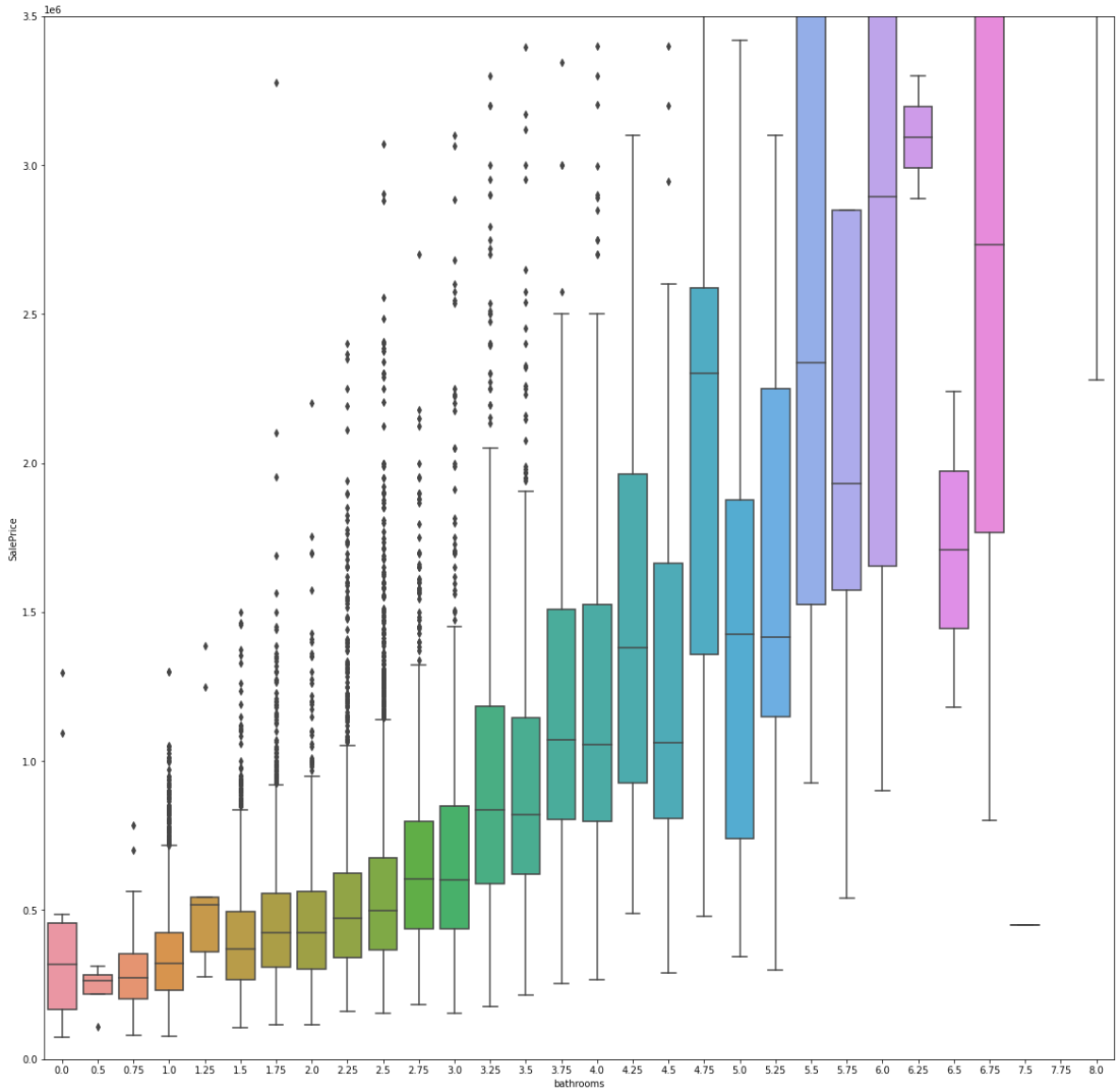
In [21]:

```
var = 'grade'
data = pd.concat([df_train['SalePrice'], df_train[var]], axis=1)
f, ax = plt.subplots(figsize=(14, 6))
fig = sns.violinplot(x=var, y="SalePrice", data=data)
fig.axis(ymin=0, ymax=3500000);
```



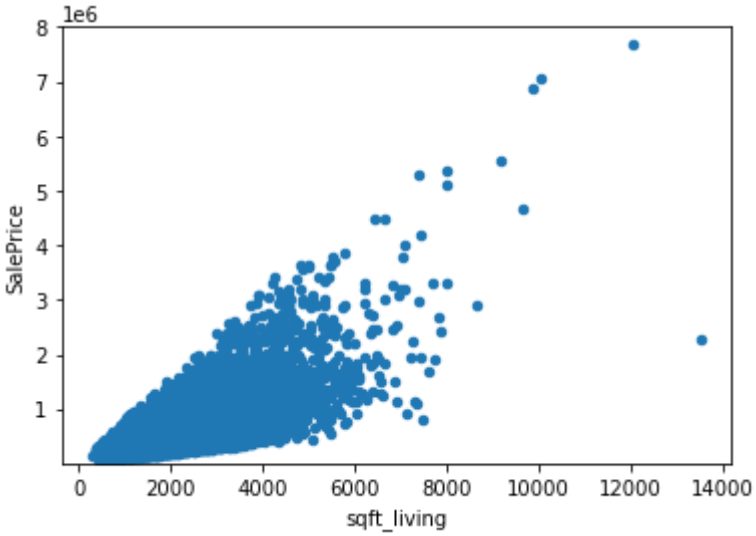
In [22]:

```
var = 'bathrooms'
data = pd.concat([df_train['SalePrice'], df_train[var]], axis=1)
f, ax = plt.subplots(figsize=(20, 20))
fig = sns.boxplot(x=var, y="SalePrice", data=data)
fig.axis(ymin=0, ymax=3500000);
```



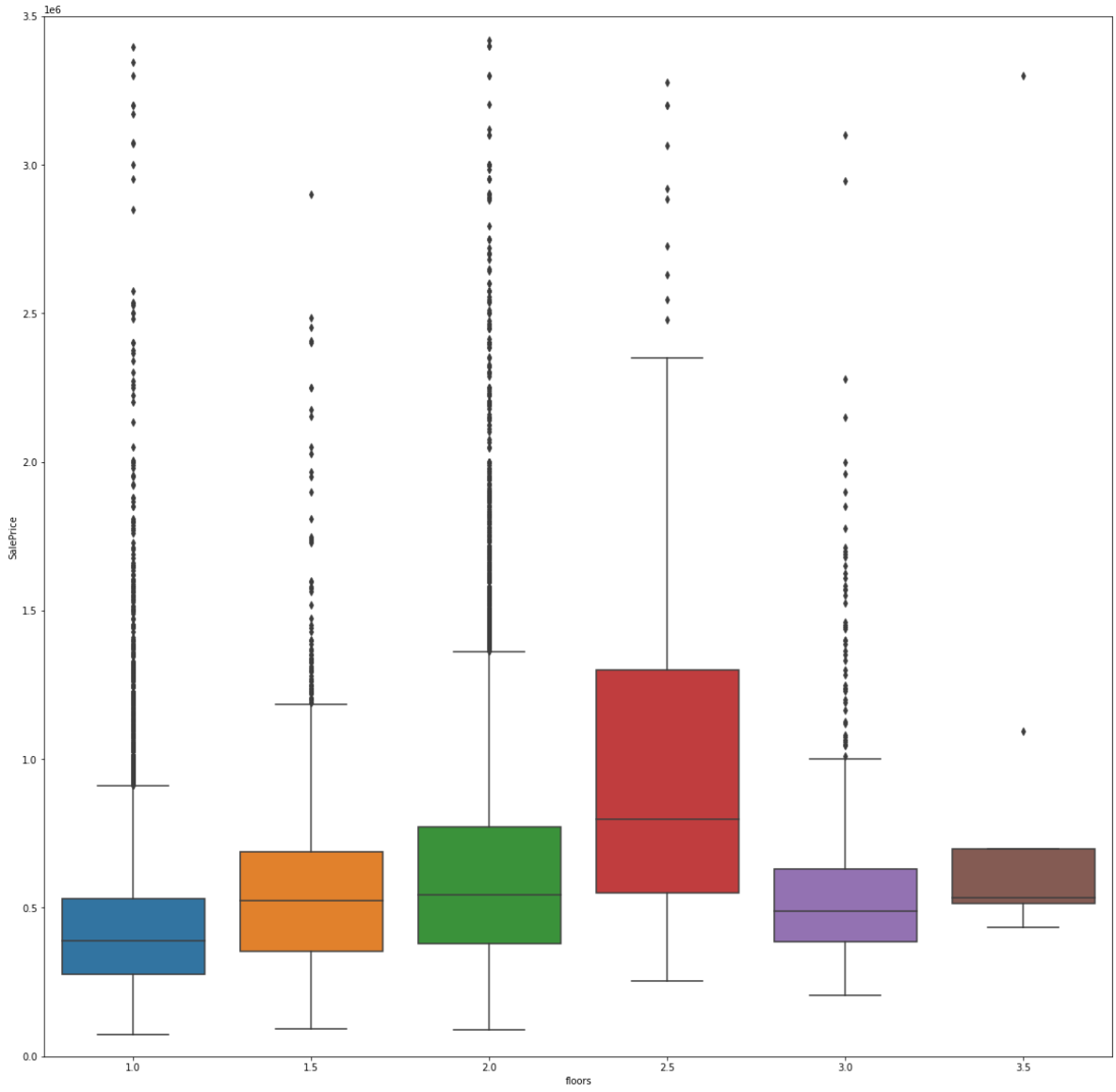
In [23]:

```
var = 'sqft_living'
data = pd.concat([df_train['SalePrice'], df_train[var]], axis=1)
data.plot.scatter(x=var, y='SalePrice', ylim=(3,8000000));
```



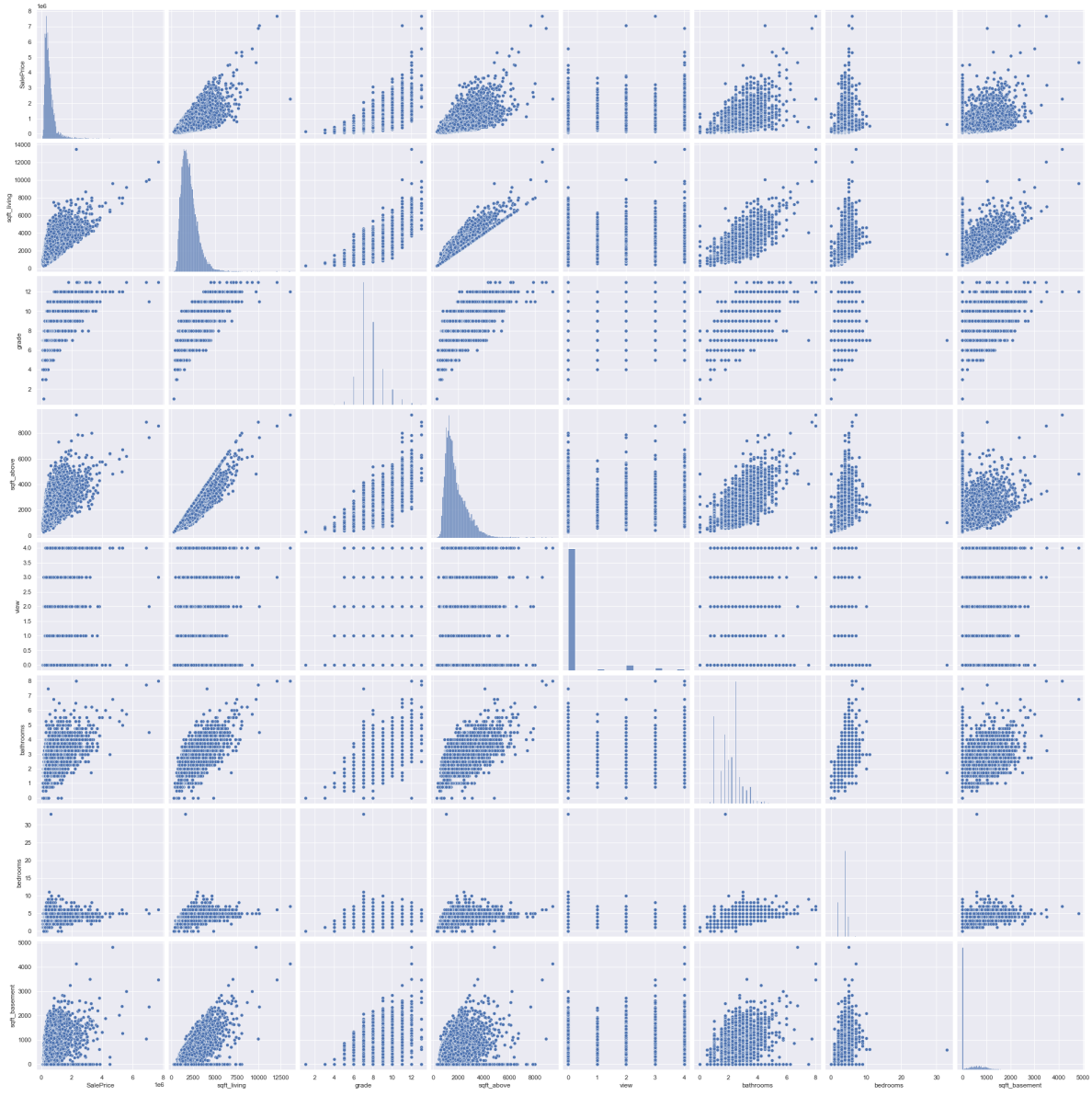
In [24]:

```
var = 'floors'
data = pd.concat([df_train['SalePrice'], df_train[var]], axis=1)
f, ax = plt.subplots(figsize=(20, 20))
fig = sns.boxplot(x=var, y="SalePrice", data=data)
fig.axis(ymin=0, ymax=3500000);
```



In [25]:

```
#Pairplots to visualize strong correlation
sns.set()
cols = ['SalePrice', 'sqft_living', 'grade', 'sqft_above', 'view', 'bathrooms','bedrooms','
sns.pairplot(df_train[cols], height = 3.5)
plt.show();
```



In [27]:

```
filtered_data = df_train[['sqft_living','grade', 'sqft_above', 'sqft_living15','bathrooms',
```

In [28]:

```
X = filtered_data.values
y = df_train.SalePrice.values
```

STEP 5 : SPLITTING DATA INTO TRAINING AND TESTING SET

In [31]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y ,test_size=0.2)
```

STEP 6: APPLYING MACHINE LEARNING MODEL

Adaboost

In [32]:

```
from sklearn.ensemble import AdaBoostRegressor
ada=AdaBoostRegressor(n_estimators=50, learning_rate=0.2,loss='exponential').fit(X_train, y
pred=ada.predict(X_test)
adab=ada.score(X_test,y_test)
predict = ada.predict(X_test)
exp_ada = explained_variance_score(predict,y_test)
```

GBM

In [33]:

```
from sklearn.ensemble import GradientBoostingRegressor
gbm=GradientBoostingRegressor(n_estimators=400, max_depth=5, loss='ls',min_samples_split=2,
gradient=gbm.score(X_test,y_test)

pred = gbm.predict(X_test)
exp_est = explained_variance_score(pred,y_test)
```

In [34]:

```
# Comparing Models on the basis of Model's Accuracy Score and Explained Variance Score of d
models_cross = pd.DataFrame({
    'Model': ['Gradient Boosting','AdaBoost'],
    'Score': [gradient,adab],
     'Variance Score': [exp_est,exp_ada]})

models_cross.sort_values(by='Score', ascending=False)
```

Out[34]:

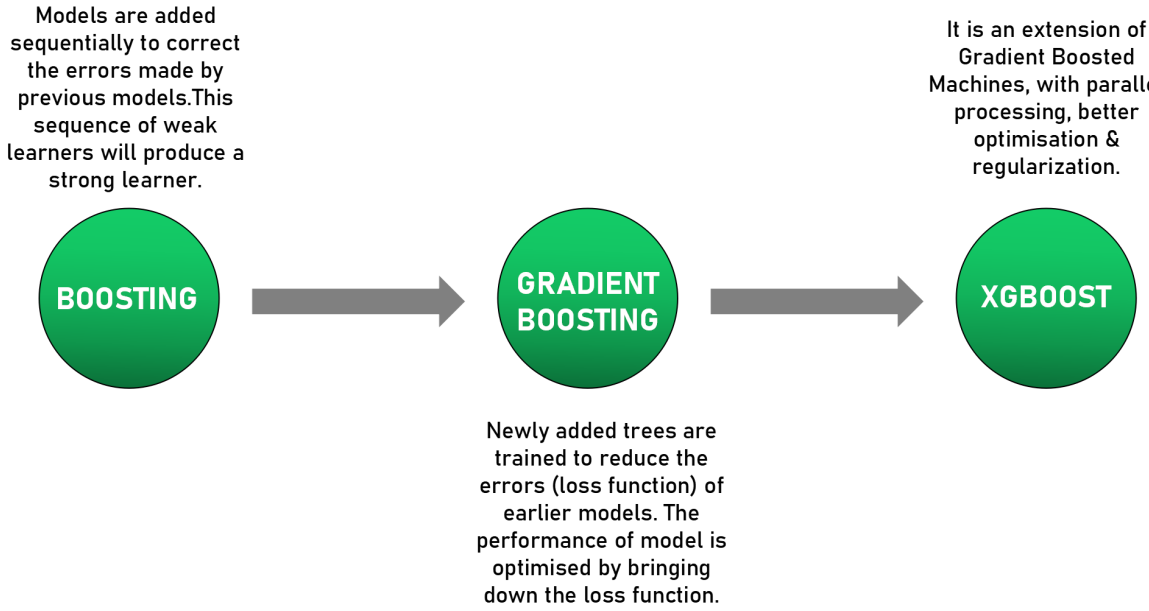
	Model	Score	Variance Score
0	Gradient Boosting	0.888402	0.879083
1	AdaBoost	0.655424	0.518544

Understanding XGBoost

Extreme Gradient Boosting (**XGBoost**) is similar to the gradient boosting framework but more efficient and advanced implementation of the Gradient Boosting algorithm.

It was first developed by Taiqi Chen and became famous in solving the Higgs Boson problem. Due to its robust accuracy, it has been widely used in machine learning competitions as well.

In the next video, with this understanding, let's see how does XGBoost works



- **AdaBoost** is an iterative way of adding weak learners to form the final model. For this, each model is trained to correct the errors made by the previous one. This sequential model does this by adding more weight to cases with incorrect predictions. By this approach, the ensemble model will correct itself while learning by focusing on cases/data points that are hard to predict correctly.
- Next, we will talk about **Gradient Boosting**. You learned about gradient descent in the earlier module. The same principle applies here as well, where the newly added trees are trained to **reduce the errors** (**loss function**) of earlier models. So, overall, in gradient boosting, we are optimizing the performance of the boosted model by bringing down the loss one small step at a time.
- **XGBoost** is an extended version of **gradient boosting**, where it uses more accurate approximations to tune the model and find the best fit.

2. **Regularization:** The biggest advantage of xgboost is that it uses regularization and controls the overfitting and simplicity of the model which gives it better performance.
3. **Enabled Cross-Validation:** XGBoost is enabled with internal Cross Validation function
4. **Missing Values:** XGBoost is designed to handle missing values internally. The missing values are treated in such a manner that if there exists any trend in missing values, it is captured by the model.
5. **Flexibility:** XGBoost is not just limited to regression, classification, and ranking problems, it supports user-defined objective functions as well. Furthermore, it supports user-defined evaluation metrics as well.

Because of parallel processing (speed) and model performance, we can say that XGBoost is gradient boosting on steroids.

Question 1/2 Mandatory

Features of XGBoost

Which of the following is/are a mandatory data pre-processing step(s) for XGBOOST?

☐ Impute Missing Values ✕ Incorrect

Remove highly correlated predictors. ✗ Incorrect

Feedback:
Decision trees are by nature immune to multicollinearity. Since boosted trees use individual decision trees, they also are unaffected by multicollinearity.

☐ One-hot encoding or dummy variable creation of categorical predictors ✓ Correct You missed this!

Feedback:
Yes converting categorical data to numeric format is required

☐ Normalization and scaling ✗ Incorrect

Feedback:
For the decision tree algorithm, normalization is not necessary in the training model procedure.

Your answer is Wrong. Attempt 2 of 2 Continue

Now, that you have gone through the basic intuition behind XGBoost, let's study how to implement it practically.

Report an error

2. **Regularization:** The biggest advantage of xgboost is that it uses regularization and controls the overfitting and simplicity of the model which gives it better performance.
3. **Enabled Cross-Validation:** XGBoost is enabled with internal Cross Validation function
4. **Missing Values:** XGBoost is designed to handle missing values internally. The missing values are treated in such a manner that if there exists any trend in missing values, it is captured by the model.
5. **Flexibility:** XGBoost is not just limited to regression, classification, and ranking problems, it supports user-defined objective functions as well. Furthermore, it supports user-defined evaluation metrics as well.

Because of parallel processing (speed) and model performance, we can say that XGBoost is gradient boosting on steroids.

Question 2/2 Mandatory

True/False

The major difference between the Gradient Boosting and the XGBoost is that XGBoost incorporates the regularisation parameter in its objective function to control over-fitting

○ True ✓ Correct

XGBoost Lab

The objective of this segment is to learn how to implement the XGBoost algorithm in python.
But before that, let's look at some of the hyperparameters used in XGBoost.

Hyperparameters - Learning Rate, Number of Trees and Subsampling

λ_t, the **learning rate,** is also known as **shrinkage.** It can be used to regularize the gradient tree boosting algorithm. λ_t typically varies from 0 to 1. Smaller values of λ_t lead to a larger value of a number of trees T (called *n_estimators* in the Python package XGBoost). This is because, with a slower learning rate, you need a larger number of trees to reach the minima. This, in turn, leads to longer training time.

On the other hand, if λ_t is large, we may reach the same point with a lesser number of trees (*n_estimators*), but there's the risk that we might actually miss the minima altogether (i.e. cross over it) because of the long stride we are taking at each iteration.

Some other ways of regularization are explicitly specifying the **number of trees T** and doing **subsampling**. Note that you shouldn't tune both λ_t and number of trees T together since a high λ_t implies a low value of T and vice-versa.

therefore look similar. This is not a big problem in boosting since each tree is any way built on the residual and gets a significantly different objective function than the previous one.

γ ,**Gamma** is a parameter used for controlling the pruning of the tree. A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split and makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.

Apart from the above-mentioned hyperparameters, there are other parameters of decision trees like the depth of the tree, the minimum number of samples required for split etc.

NOTE: The mathematical understanding behind XGboost is provided in the optional section.
For a detailed understanding, please attend the live session on Saturday.

Question 1/2 Mandatory

Hyperparameter Tuning in XGBoost

The hyperparameters λ_t (learning rate) and T (number of trees):

Feedback:
The faster the learning rate, the faster you move towards the minima, thus requiring a lower number of trees.

Are dependent on each other - the faster the learning rate, the higher the number of trees ✕ Incorrect

Feedback:
The faster the learning rate, the faster you move towards the minima, thus requiring a lower number of trees.

Your answer is Wrong. Attempt 1 of 1 Continue

With all this understanding let's move to the next video where Snehanshu will give you a walkthrough on how to do classification with XGBoost.

XGBoost - classification Download

Now, that you have gone through classification, let's dive into the regression problem. Snehanshu will now explain how we approach regression problem wiith XGBoost through a code walkthrough.

XGBoost - regression Download

Download and implement the code in the following notebook to get an understanding of both XGBoost classification & regression.

Report an error

PREVIOUS
Understanding XGBoost

NEXT
Kaggle Practice Exercise

therefore look similar. This is not a big problem in boosting since each tree is any way built on the residual and gets a significantly different objective function than the previous one.

γ ,**Gamma** is a parameter used for controlling the pruning of the tree. A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split and makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.

Apart from the above-mentioned hyperparameters, there are other parameters of decision trees like the depth of the tree, the minimum number of samples required for split etc.

NOTE: The mathematical understanding behind XGboost is provided in the optional section.
For a detailed understanding, please attend the live session on Saturday.

Question 2/2 Mandatory

XGBoost Regularization

What will happen if we increase the regularisation parameter λ_t?

The number of trees will increase

Your answer is Correct. Attempt 1 of 1 Continue

With all this understanding let's move to the next video where Snehanshu will give you a walkthrough on how to do classification with XGBoost.

In [1]:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer

pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
```

In [2]:

```
# Read the Dataset

data = load_breast_cancer()
data
```

Out[2]:

```
{'data': array([[1.799e+01, 1.038e+01, 1.228e+02, ..., 2.654e-01, 4.601e-01,
         1.189e-01],
        [2.057e+01, 1.777e+01, 1.329e+02, ..., 1.860e-01, 2.750e-01,
         8.902e-02],
        [1.969e+01, 2.125e+01, 1.300e+02, ..., 2.430e-01, 3.613e-01,
         8.758e-02],
        ...,
        [1.660e+01, 2.808e+01, 1.083e+02, ..., 1.418e-01, 2.218e-01,
         7.820e-02],
        [2.060e+01, 2.933e+01, 1.401e+02, ..., 2.650e-01, 4.087e-01,
         1.240e-01],
        [7.760e+00, 2.454e+01, 4.792e+01, ..., 0.000e+00, 2.871e-01,
         7.039e-02]]),
 'target': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 1, 1,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
        0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0,
        1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
        1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1,
        1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0,
        0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1,
        1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0,
        0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0,
        1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1,
        1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0,
        0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0,
        0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1,
        1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1,
        1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
        1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
        1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1,
        1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1]),
 'frame': None,
 'target_names': array(['malignant', 'benign'], dtype='<U9'),
 'DESCR': '.. _breast_cancer_dataset:\n\nBreast cancer wisconsin (diagnosti
c) dataset\n--------------------------------------------\n\n**Data Set Chara
cteristics:**\n\n    :Number of Instances: 569\n\n    :Number of Attributes:
30 numeric, predictive attributes and the class\n\n    :Attribute Informatio
n:\n        - radius (mean of distances from center to points on the perimet
er)\n        - texture (standard deviation of gray-scale values)\n        -
perimeter\n        - area\n        - smoothness (local variation in radius l
engths)\n        - compactness (perimeter^2 / area - 1.0)\n        - concavi
ty (severity of concave portions of the contour)\n        - concave points
(number of concave portions of the contour)\n        - symmetry\n        - f
```

ractal dimension ("coastline approximation" - 1)\n\n The mean, standa
rd error, and "worst" or largest (mean of the three\n worst/largest v
alues) of these features were computed for each image,\n resulting in
30 features. For instance, field 0 is Mean Radius, field\n 10 is Rad
ius SE, field 20 is Worst Radius.\n\n - class:\n - WDB
C-Malignant\n - WDBC-Benign\n\n :Summary Statistics:\n\n
===================================== ====== ======\n
Min Max\n ===================================== ====== ======\n rad
ius (mean): 6.981 28.11\n texture (mean):
9.71 39.28\n perimeter (mean): 43.79 188.5\n ar
ea (mean): 143.5 2501.0\n smoothness (mean):
0.053 0.163\n compactness (mean): 0.019 0.345\n co
ncavity (mean): 0.0 0.427\n concave points (mean):
0.0 0.201\n symmetry (mean): 0.106 0.304\n fr
actal dimension (mean): 0.05 0.097\n radius (standard erro
r): 0.112 2.873\n texture (standard error): 0.3
6 4.885\n perimeter (standard error): 0.757 21.98\n area
(standard error): 6.802 542.2\n smoothness (standard erro
r): 0.002 0.031\n compactness (standard error): 0.002
0.135\n concavity (standard error): 0.0 0.396\n concave p
oints (standard error): 0.0 0.053\n symmetry (standard error):
0.008 0.079\n fractal dimension (standard error): 0.001 0.03\n rad
ius (worst): 7.93 36.04\n texture (worst):
12.02 49.54\n perimeter (worst): 50.41 251.2\n ar
ea (worst): 185.2 4254.0\n smoothness (worst):
0.071 0.223\n compactness (worst): 0.027 1.058\n co
ncavity (worst): 0.0 1.252\n concave points (wors
t): 0.0 0.291\n symmetry (worst): 0.
156 0.664\n fractal dimension (worst): 0.055 0.208\n ====
=================================== ====== ======\n\n :Missing Attribute Va
lues: None\n\n :Class Distribution: 212 - Malignant, 357 - Benign\n\n
:Creator: Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian\n\n
:Donor: Nick Street\n\n :Date: November, 1995\n\nThis is a copy of UCI ML
Breast Cancer Wisconsin (Diagnostic) datasets.\nhttps://goo.gl/U2Uwz2\n\nFea
tures are computed from a digitized image of a fine needle\naspirate (FNA) o
f a breast mass. They describe\ncharacteristics of the cell nuclei present
in the image.\n\nSeparating plane described above was obtained using\nMultis
urface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree\nConstruction Via
Linear Programming." Proceedings of the 4th\nMidwest Artificial Intelligence
and Cognitive Science Society,\npp. 97-101, 1992], a classification method w
hich uses linear\nprogramming to construct a decision tree. Relevant featur
es\nwere selected using an exhaustive search in the space of 1-4\nfeatures a
nd 1-3 separating planes.\n\nThe actual linear program used to obtain the se
parating plane\nin the 3-dimensional space is that described in:\n[K. P. Ben
nett and O. L. Mangasarian: "Robust Linear\nProgramming Discrimination of Tw
o Linearly Inseparable Sets",\nOptimization Methods and Software 1, 1992, 23
-34].\n\nThis database is also available through the UW CS ftp server:\n\nft
p ftp.cs.wisc.edu\ncd math-prog/cpo-dataset/machine-learn/WDBC/\n\n.. topi
c:: References\n\n - W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nucle
ar feature extraction \n for breast tumor diagnosis. IS&T/SPIE 1993 Inte
rnational Symposium on \n Electronic Imaging: Science and Technology, vo
lume 1905, pages 861-870,\n San Jose, CA, 1993.\n - O.L. Mangasarian,
W.N. Street and W.H. Wolberg. Breast cancer diagnosis and \n prognosis v
ia linear programming. Operations Research, 43(4), pages 570-577, \n Jul
y-August 1995.\n - W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machin
e learning techniques\n to diagnose breast cancer from fine-needle aspir
ates. Cancer Letters 77 (1994) \n 163-171.',
'feature_names': array(['mean radius', 'mean texture', 'mean perimeter', 'm
ean area',
'mean smoothness', 'mean compactness', 'mean concavity',
'mean concave points', 'mean symmetry', 'mean fractal dimension',

```
        'radius error', 'texture error', 'perimeter error', 'area error',
        'smoothness error', 'compactness error', 'concavity error',
        'concave points error', 'symmetry error',
        'fractal dimension error', 'worst radius', 'worst texture',
        'worst perimeter', 'worst area', 'worst smoothness',
        'worst compactness', 'worst concavity', 'worst concave points',
        'worst symmetry', 'worst fractal dimension'], dtype='<U23'),
 'filename': '/Users/ssahu/anaconda3.8.3_2020.07_cf/lib/python3.8/site-packa
ges/sklearn/datasets/data/breast_cancer.csv'}
```

In [3]:

```
print(data.DESCR)
```

```
.. _breast_cancer_dataset:

Breast cancer wisconsin (diagnostic) dataset
--------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 569

    :Number of Attributes: 30 numeric, predictive attributes and the clas
s

    :Attribute Information:
        - radius (mean of distances from center to points on the perimete
r)
        - texture (standard deviation of gray-scale values)
        - perimeter
        - area
        - smoothness (local variation in radius lengths)
        - compactness (perimeter^2 / area - 1.0)
```

In [4]:

```
print(type(data['data']))
print(type(data['target']))
```

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
```

In [5]:

```
print(data['data'].shape)
print(data['target'].shape)
print(data['target'][data['target']==0].shape)
print(data['target'][data['target']==1].shape)
```

```
(569, 30)
(569,)
(212,)
(357,)
```

In [6]:

```
df = pd.DataFrame(data.data, columns=data.feature_names)
df
```

Out[6]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	me frac dimensi
0	17.990	10.38	122.80	1001.0	0.11840	0.27760	0.300100	0.147100	0.2419	0.078
1	20.570	17.77	132.90	1326.0	0.08474	0.07864	0.086900	0.070170	0.1812	0.056
2	19.690	21.25	130.00	1203.0	0.10960	0.15990	0.197400	0.127900	0.2069	0.059
3	11.420	20.38	77.58	386.1	0.14250	0.28390	0.241400	0.105200	0.2597	0.097
4	20.290	14.34	135.10	1297.0	0.10030	0.13280	0.198000	0.104300	0.1809	0.058
5	12.450	15.70	82.57	477.1	0.12780	0.17000	0.157800	0.080890	0.2087	0.076
6	18.250	19.98	119.60	1040.0	0.09463	0.10900	0.112700	0.074000	0.1794	0.057
7	13.710	20.83	90.20	577.9	0.11890	0.16450	0.093660	0.059850	0.2196	0.074
8	13.000	21.82	87.50	519.8	0.12730	0.19320	0.185900	0.093530	0.2350	0.073

In [7]:

```
df['CANCER'] = data.target
df
```

Out[7]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	me frac dimensi
0	17.990	10.38	122.80	1001.0	0.11840	0.27760	0.300100	0.147100	0.2419	0.078
1	20.570	17.77	132.90	1326.0	0.08474	0.07864	0.086900	0.070170	0.1812	0.056
2	19.690	21.25	130.00	1203.0	0.10960	0.15990	0.197400	0.127900	0.2069	0.059
3	11.420	20.38	77.58	386.1	0.14250	0.28390	0.241400	0.105200	0.2597	0.097
4	20.290	14.34	135.10	1297.0	0.10030	0.13280	0.198000	0.104300	0.1809	0.058
5	12.450	15.70	82.57	477.1	0.12780	0.17000	0.157800	0.080890	0.2087	0.076
6	18.250	19.98	119.60	1040.0	0.09463	0.10900	0.112700	0.074000	0.1794	0.057
7	13.710	20.83	90.20	577.9	0.11890	0.16450	0.093660	0.059850	0.2196	0.074
8	13.000	21.82	87.50	519.8	0.12730	0.19320	0.185900	0.093530	0.2350	0.073

In [8]:

```
df.describe(include='all')
```

Out[8]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000
mean	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720
min	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000
25%	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560
50%	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540
75%	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800

In [9]:

```
df.info()
```

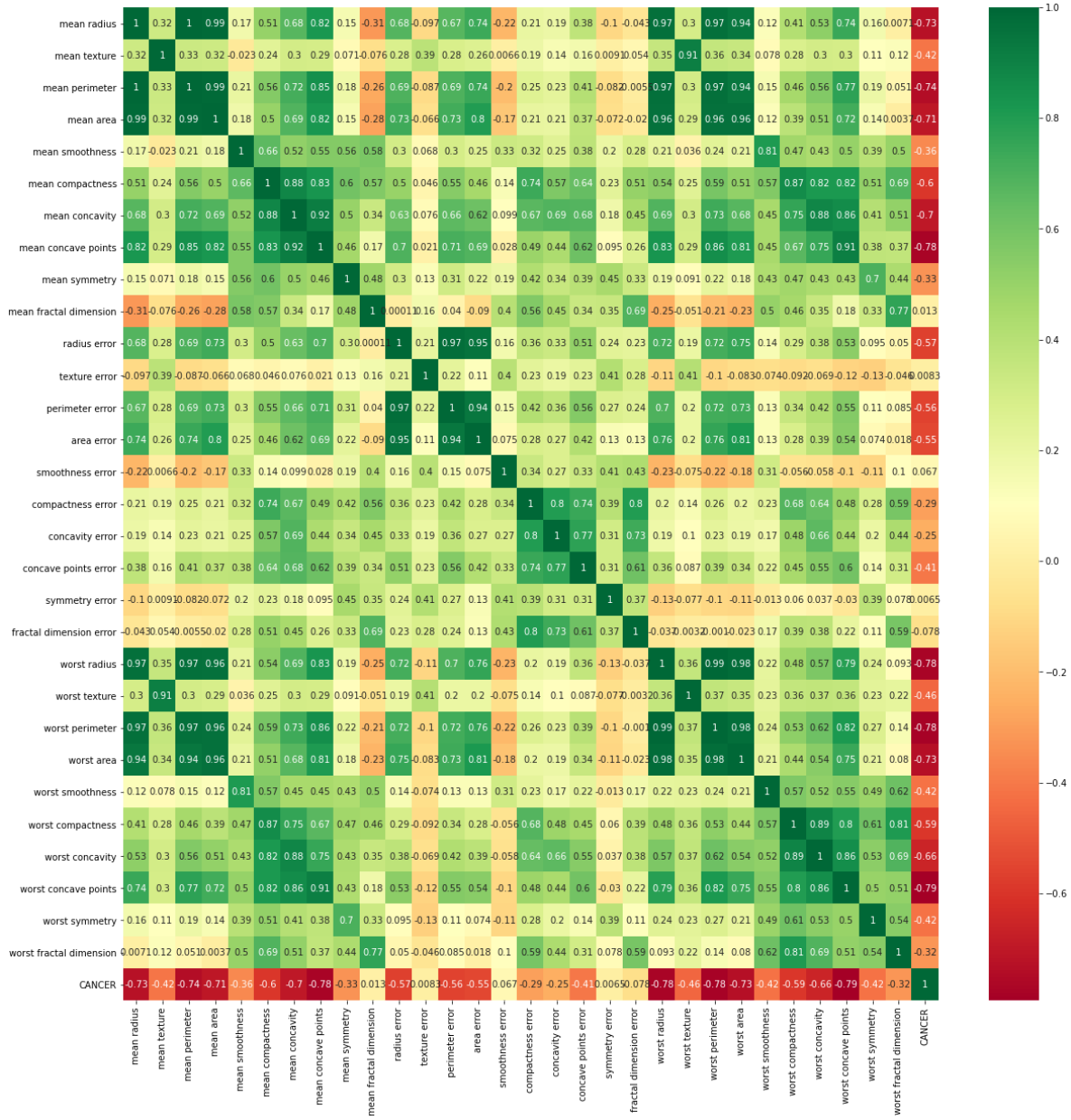
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 31 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   mean radius              569 non-null    float64
 1   mean texture             569 non-null    float64
 2   mean perimeter           569 non-null    float64
 3   mean area                569 non-null    float64
 4   mean smoothness          569 non-null    float64
 5   mean compactness         569 non-null    float64
 6   mean concavity           569 non-null    float64
 7   mean concave points      569 non-null    float64
 8   mean symmetry            569 non-null    float64
 9   mean fractal dimension   569 non-null    float64
 10  radius error             569 non-null    float64
 11  texture error            569 non-null    float64
 12  perimeter error          569 non-null    float64
 13  area error               569 non-null    float64
 14  smoothness error         569 non-null    float64
 15  compactness error        569 non-null    float64
 16  concavity error          569 non-null    float64
 17  concave points error     569 non-null    float64
 18  symmetry error           569 non-null    float64
 19  fractal dimension error  569 non-null    float64
 20  worst radius             569 non-null    float64
 21  worst texture            569 non-null    float64
 22  worst perimeter          569 non-null    float64
 23  worst area               569 non-null    float64
 24  worst smoothness         569 non-null    float64
 25  worst compactness        569 non-null    float64
 26  worst concavity          569 non-null    float64
 27  worst concave points     569 non-null    float64
 28  worst symmetry           569 non-null    float64
 29  worst fractal dimension  569 non-null    float64
 30  CANCER                   569 non-null    int64  
dtypes: float64(30), int64(1)
memory usage: 137.9 KB
```

In [10]:

```
# Correlation

import seaborn as sns
import matplotlib.pyplot as plt

# get correlations of each pair of features in the data
corrmat = df.corr()
top_corr_features = corrmat.index
plt.figure(figsize=(20, 20))

# plot heat map
sns.heatmap(df[top_corr_features].corr(), annot=True, cmap="RdYlGn")
```

Out[10]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x119f2b850>
```



In [11]:

```
X, y = df.iloc[:, :-1], df.iloc[:, -1]
```

In [12]:

```
print(X.shape)
print(y.shape)
```

```
(569, 30)
(569,)
```

In [13]:

```
from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(X,
                                                                    y,
                                                                    test_size=0.2,
                                                                    stratify=y,
                                                                    shuffle=True,
                                                                    random_state=0)
print(y.shape[0])
print(y_train.shape[0])
print(y_test.shape[0])
```

```
569
455
114
```

In [14]:

```
print(y_train.shape[0])
print(y_train[y_train==0].shape)
print(y_train[y_train==1].shape)
```

```
455
(170,)
(285,)
```

In [15]:

```
print(y_test.shape[0])
print(y_test[y_test==0].shape)
print(y_test[y_test==1].shape)
```

```
114
(42,)
(72,)
```

In [16]:

```
import xgboost as xgb
from sklearn import metrics

xgclf = xgb.XGBClassifier()

xgclf.fit(X_train, y_train)
```

Out[16]:

```
XGBClassifier(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=6,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=100, n_jobs=0, num_parallel_tree=1, random_state=
1,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=None)
```

In [17]:

```
print('AUC on train data by XGBoost =', metrics.roc_auc_score(y_true=y_train,
                                                              y_score=xgclf.predict_proba(X
```

```
AUC on train data by XGBoost = 1.0
```

In [18]:

```
print('AUC on test data by XGBoost =', metrics.roc_auc_score(y_true=y_test,
                                                             y_score=xgclf.predict_proba(X_
```

```
AUC on test data by XGBoost = 0.9887566137566137
```

In [19]:

```
xgb_model = xgb.XGBClassifier()

# Default-Run of default-hyperparameters
parameters = {'learning_rate': [0.3],
              'max_depth': [6],
              'min_child_weight': [1],
              'n_estimators': [100]}

scorer = metrics.make_scorer(metrics.roc_auc_score,
                             greater_is_better=True,
                             needs_proba=True,
                             needs_threshold=False)

clf_xgb = model_selection.GridSearchCV(estimator=xgb_model,
                                       param_grid=parameters,
                                       n_jobs=-1,
                                       cv=3,
                                       scoring=scorer,
                                       refit=True)

clf_xgb.fit(X_train, y_train)
```

Out[19]:

```
GridSearchCV(cv=3,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None, gamma=None,
                                     gpu_id=None, importance_type='gain',
                                     interaction_constraints=None,
                                     learning_rate=None, max_delta_step=Non
e,
                                     max_depth=None, min_child_weight=None,
                                     missing=nan, monotone_constraints=None,
                                     n_estimators=100, n_jobs=None,
                                     num_parallel_tree=None, random_state=No
ne,
                                     reg_alpha=None, reg_lambda=None,
                                     scale_pos_weight=None, subsample=None,
                                     tree_method=None, validate_parameters=N
one,
                                     verbosity=None),
             n_jobs=-1,
             param_grid={'learning_rate': [0.3], 'max_depth': [6],
                         'min_child_weight': [1], 'n_estimators': [100]},
             scoring=make_scorer(roc_auc_score, needs_proba=True))
```

In [20]:

```
print(clf_xgb.best_params_)
print(clf_xgb.best_score_)
print(clf_xgb.best_estimator_)
```

```
{'learning_rate': 0.3, 'max_depth': 6, 'min_child_weight': 1, 'n_estimator
s': 100}
0.991775491359979
XGBClassifier(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=0.3, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=100, n_jobs=0, num_parallel_tree=1, random_state=
0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=None)
```

In [21]:

```
# 1st-Run for best hyperparameters
parameters = {'learning_rate': [0.1, 0.2, 0.3, 0.4, 0.5],
              'max_depth': [2, 4, 6, 8, 10],
              'min_child_weight': [3, 7, 11, 19, 25],
              'n_estimators': [50, 100, 150, 200, 300, 500]}

scorer = metrics.make_scorer(metrics.roc_auc_score,
                             greater_is_better=True,
                             needs_proba=True,
                             needs_threshold=False)

clf_xgb = model_selection.GridSearchCV(estimator=xgb_model,
                                       param_grid=parameters,
                                       n_jobs=-1,
                                       cv=3,
                                       scoring=scorer,
                                       refit=True)

clf_xgb.fit(X_train, y_train)
```

Out[21]:

```
GridSearchCV(cv=3,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None, gamma=None,
                                     gpu_id=None, importance_type='gain',
                                     interaction_constraints=None,
                                     learning_rate=None, max_delta_step=Non
e,
                                     max_depth=None, min_child_weight=None,
                                     missing=nan, monotone_constraints=None,
                                     n_estimators=100, n_jobs...
                                     num_parallel_tree=None, random_state=No
ne,
                                     reg_alpha=None, reg_lambda=None,
                                     scale_pos_weight=None, subsample=None,
                                     tree_method=None, validate_parameters=N
one,
                                     verbosity=None),
             n_jobs=-1,
             param_grid={'learning_rate': [0.1, 0.2, 0.3, 0.4, 0.5],
                         'max_depth': [2, 4, 6, 8, 10],
                         'min_child_weight': [3, 7, 11, 19, 25],
                         'n_estimators': [50, 100, 150, 200, 300, 500]},
             scoring=make_scorer(roc_auc_score, needs_proba=True))
```

In [22]:

```
print(clf_xgb.best_params_)
print(clf_xgb.best_score_)
print(clf_xgb.best_estimator_)
```

```
{'learning_rate': 0.2, 'max_depth': 4, 'min_child_weight': 3, 'n_estimator
s': 100}
0.9915842237171878
XGBClassifier(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=0.2, max_delta_step=0, max_depth=4,
              min_child_weight=3, missing=nan, monotone_constraints='()',
              n_estimators=100, n_jobs=0, num_parallel_tree=1, random_state=
0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=None)
```

In [23]:

```
final_model = xgb.XGBClassifier(learning_rate=0.3,
                                max_depth=6,
                                min_child_weight=1,
                                n_estimators=100)

final_model.fit(X_train, y_train)
print('AUC on train data by XGBoost =', metrics.roc_auc_score(y_true=y_train,
                                                              y_score=final_model.predict_p

print('AUC on test data by XGBoost =', metrics.roc_auc_score(y_true=y_test,
                                                             y_score=final_model.predict_pr
```

```
AUC on train data by XGBoost = 1.0
AUC on test data by XGBoost = 0.9887566137566137
```

Hyperparameter Optimization For XGBoost using GridSearchCV

In [1]:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_boston

pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
```

In [2]:

```
# Read the Dataset

data = load_boston()
data
```

Out[2]:

```
{'data': array([[6.3200e-03, 1.8000e+01, 2.3100e+00, ..., 1.5300e+01, 3.9690
e+02,
         4.9800e+00],
        [2.7310e-02, 0.0000e+00, 7.0700e+00, ..., 1.7800e+01, 3.9690e+02,
         9.1400e+00],
        [2.7290e-02, 0.0000e+00, 7.0700e+00, ..., 1.7800e+01, 3.9283e+02,
         4.0300e+00],
        ...,
        [6.0760e-02, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9690e+02,
         5.6400e+00],
        [1.0959e-01, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9345e+02,
         6.4800e+00],
        [4.7410e-02, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9690e+02,
         7.8800e+00]]),
 'target': array([24. , 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.
9, 15. ,
        18.9, 21.7, 20.4, 18.2, 19.9, 23.1, 17.5, 20.2, 18.2, 13.6, 19.6,
        15.2, 14.5, 15.6, 13.9, 16.6, 14.8, 18.4, 21. , 12.7, 14.5, 13.2,
        13.1, 13.5, 18.9, 20. , 21. , 24.7, 30.8, 34.9, 26.6, 25.3, 24.7,
        21.2, 19.3, 20. , 16.6, 14.4, 19.4, 19.7, 20.5, 25. , 23.4, 18.9,
        35.4, 24.7, 31.6, 23.3, 19.6, 18.7, 16. , 22.2, 25. , 33. , 23.5,
        19.4, 22. , 17.4, 20.9, 24.2, 21.7, 22.8, 23.4, 24.1, 21.4, 20. ,
        20.8, 21.2, 20.3, 28. , 23.9, 24.8, 22.9, 23.9, 26.6, 22.5, 22.2,
        23.6, 28.7, 22.6, 22. , 22.9, 25. , 20.6, 28.4, 21.4, 38.7, 43.8,
        33.2, 27.5, 26.5, 18.6, 19.3, 20.1, 19.5, 19.5, 20.4, 19.8, 19.4,
        21.7, 22.8, 18.8, 18.7, 18.5, 18.3, 21.2, 19.2, 20.4, 19.3, 22. ,
        20.3, 20.5, 17.3, 18.8, 21.4, 15.7, 16.2, 18. , 14.3, 19.2, 19.6,
        23. , 18.4, 15.6, 18.1, 17.4, 17.1, 13.3, 17.8, 14. , 14.4, 13.4,
        15.6, 11.8, 13.8, 15.6, 14.6, 17.8, 15.4, 21.5, 19.6, 15.3, 19.4,
        17. , 15.6, 13.1, 41.3, 24.3, 23.3, 27. , 50. , 50. , 50. , 22.7,
        25. , 50. , 23.8, 23.8, 22.3, 17.4, 19.1, 23.1, 23.6, 22.6, 29.4,
        23.2, 24.6, 29.9, 37.2, 39.8, 36.2, 37.9, 32.5, 26.4, 29.6, 50. ,
        32. , 29.8, 34.9, 37. , 30.5, 36.4, 31.1, 29.1, 50. , 33.3, 30.3,
        34.6, 34.9, 32.9, 24.1, 42.3, 48.5, 50. , 22.6, 24.4, 22.5, 24.4,
        20. , 21.7, 19.3, 22.4, 28.1, 23.7, 25. , 23.3, 28.7, 21.5, 23. ,
        26.7, 21.7, 27.5, 30.1, 44.8, 50. , 37.6, 31.6, 46.7, 31.5, 24.3,
        31.7, 41.7, 48.3, 29. , 24. , 25.1, 31.5, 23.7, 23.3, 22. , 20.1,
        22.2, 23.7, 17.6, 18.5, 24.3, 20.5, 24.5, 26.2, 24.4, 24.8, 29.6,
        42.8, 21.9, 20.9, 44. , 50. , 36. , 30.1, 33.8, 43.1, 48.8, 31. ,
        36.5, 22.8, 30.7, 50. , 43.5, 20.7, 21.1, 25.2, 24.4, 35.2, 32.4,
        32. , 33.2, 33.1, 29.1, 35.1, 45.4, 35.4, 46. , 50. , 32.2, 22. ,
        20.1, 23.2, 22.3, 24.8, 28.5, 37.3, 27.9, 23.9, 21.7, 28.6, 27.1,
        20.3, 22.5, 29. , 24.8, 22. , 26.4, 33.1, 36.1, 28.4, 33.4, 28.2,
        22.8, 20.3, 16.1, 22.1, 19.4, 21.6, 23.8, 16.2, 17.8, 19.8, 23.1,
        21. , 23.8, 23.1, 20.4, 18.5, 25. , 24.6, 23. , 22.2, 19.3, 22.6,
        19.8, 17.1, 19.4, 22.2, 20.7, 21.1, 19.5, 18.5, 20.6, 19. , 18.7,
        32.7, 16.5, 23.9, 31.2, 17.5, 17.2, 23.1, 24.5, 26.6, 22.9, 24.1,
        18.6, 30.1, 18.2, 20.6, 17.8, 21.7, 22.7, 22.6, 25. , 19.9, 20.8,
        16.8, 21.9, 27.5, 21.9, 23.1, 50. , 50. , 50. , 50. , 50. , 13.8,
        13.8, 15. , 13.9, 13.3, 13.1, 10.2, 10.4, 10.9, 11.3, 12.3,  8.8,
         7.2, 10.5,  7.4, 10.2, 11.5, 15.1, 23.2,  9.7, 13.8, 12.7, 13.1,
        12.5,  8.5,  5. ,  6.3,  5.6,  7.2, 12.1,  8.3,  8.5,  5. , 11.9,
```

```
        27.9, 17.2, 27.5, 15. , 17.2, 17.9, 16.3,  7. ,  7.2,  7.5, 10.4,
         8.8,  8.4, 16.7, 14.2, 20.8, 13.4, 11.7,  8.3, 10.2, 10.9, 11. ,
         9.5, 14.5, 14.1, 16.1, 14.3, 11.7, 13.4,  9.6,  8.7,  8.4, 12.8,
        10.5, 17.1, 18.4, 15.4, 10.8, 11.8, 14.9, 12.6, 14.1, 13. , 13.4,
        15.2, 16.1, 17.8, 14.9, 14.1, 12.7, 13.5, 14.9, 20. , 16.4, 17.7,
        19.5, 20.2, 21.4, 19.9, 19. , 19.1, 19.1, 20.1, 19.9, 19.6, 23.2,
        29.8, 13.8, 13.3, 16.7, 12. , 14.6, 21.4, 23. , 23.7, 25. , 21.8,
        20.6, 21.2, 19.1, 20.6, 15.2,  7. ,  8.1, 13.6, 20.1, 21.8, 24.5,
        23.1, 19.7, 18.3, 21.2, 17.5, 16.8, 22.4, 20.6, 23.9, 22. , 11.9]),
 'feature_names': array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
'DIS', 'RAD',
        'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7'),
 'DESCR': ".. _boston_dataset:\n\nBoston house prices dataset\n----------
--------------\n\n**Data Set Characteristics:**  \n\n    :Number of Instance
s: 506 \n\n    :Number of Attributes: 13 numeric/categorical predictive. Med
ian Value (attribute 14) is usually the target.\n\n    :Attribute Informatio
n (in order):\n        - CRIM     per capita crime rate by town\n        - Z
N       proportion of residential land zoned for lots over 25,000 sq.ft.\n
- INDUS    proportion of non-retail business acres per town\n        - CHAS
Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)\n
- NOX      nitric oxides concentration (parts per 10 million)\n        - RM
average number of rooms per dwelling\n        - AGE      proportion of owner
-occupied units built prior to 1940\n        - DIS      weighted distances t
o five Boston employment centres\n        - RAD      index of accessibility
to radial highways\n        - TAX      full-value property-tax rate per $10,
000\n        - PTRATIO  pupil-teacher ratio by town\n        - B        1000
(Bk - 0.63)^2 where Bk is the proportion of blacks by town\n        - LSTAT
% lower status of the population\n        - MEDV     Median value of owner-o
ccupied homes in $1000's\n\n    :Missing Attribute Values: None\n\n    :Crea
tor: Harrison, D. and Rubinfeld, D.L.\n\nThis is a copy of UCI ML housing da
taset.\nhttps://archive.ics.uci.edu/ml/machine-learning-databases/housing/\n
\n\nThis dataset was taken from the StatLib library which is maintained at C
arnegie Mellon University.\n\nThe Boston house-price data of Harrison, D. an
d Rubinfeld, D.L. 'Hedonic\nprices and the demand for clean air', J. Enviro
n. Economics & Management,\nvol.5, 81-102, 1978.   Used in Belsley, Kuh & We
lsch, 'Regression diagnostics\n...', Wiley, 1980.   N.B. Various transformat
ions are used in the table on\npages 244-261 of the latter.\n\nThe Boston ho
use-price data has been used in many machine learning papers that address re
gression\nproblems.   \n     \n.. topic:: References\n\n   - Belsley, Kuh &
Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of
Collinearity', Wiley, 1980. 244-261.\n   - Quinlan,R. (1993). Combining Inst
ance-Based and Model-Based Learning. In Proceedings on the Tenth Internation
al Conference of Machine Learning, 236-243, University of Massachusetts, Amh
erst. Morgan Kaufmann.\n",
 'filename': '/Users/ssahu/anaconda3.8.3_2020.07_cf/lib/python3.8/site-packa
ges/sklearn/datasets/data/boston_house_prices.csv'}
```

In [3]:

```
print(data.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Valu
e (attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,
000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds rive
r; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks
by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/ (http
s://archive.ics.uci.edu/ml/machine-learning-databases/housing/)


This dataset was taken from the StatLib library which is maintained at Ca
rnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnos
tics
...', Wiley, 1980.   N.B. Various transformations are used in the table o
n
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers
that address regression
problems.

.. topic:: References
```

```
   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influent
ial Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learnin
g. In Proceedings on the Tenth International Conference of Machine Learni
ng, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
```

In [4]:

```
print(type(data['data']))
print(type(data['target']))
```

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
```

In [5]:

```
print(data['data'].shape)
print(data['target'].shape)
```

```
(506, 13)
(506,)
```

In [6]:

```
df = pd.DataFrame(data.data, columns=data.feature_names)
df
```

Out[6]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.5380	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.4690	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.4690	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.4580	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.4580	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33
5	0.02985	0.0	2.18	0.0	0.4580	6.430	58.7	6.0622	3.0	222.0	18.7	394.12	5.21
6	0.08829	12.5	7.87	0.0	0.5240	6.012	66.6	5.5605	5.0	311.0	15.2	395.60	12.43
7	0.14455	12.5	7.87	0.0	0.5240	6.172	96.1	5.9505	5.0	311.0	15.2	396.90	19.15
8	0.21124	12.5	7.87	0.0	0.5240	5.631	100.0	6.0821	5.0	311.0	15.2	386.63	29.93
9	0.17004	12.5	7.87	0.0	0.5240	6.004	85.9	6.5921	5.0	311.0	15.2	386.71	17.10
10	0.22489	12.5	7.87	0.0	0.5240	6.377	94.3	6.3467	5.0	311.0	15.2	392.52	20.45

In [7]:

```
df['PRICE'] = data.target
df
```

Out[7]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.5380	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.4690	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.4690	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.4580	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.4580	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33
5	0.02985	0.0	2.18	0.0	0.4580	6.430	58.7	6.0622	3.0	222.0	18.7	394.12	5.21
6	0.08829	12.5	7.87	0.0	0.5240	6.012	66.6	5.5605	5.0	311.0	15.2	395.60	12.43
7	0.14455	12.5	7.87	0.0	0.5240	6.172	96.1	5.9505	5.0	311.0	15.2	396.90	19.15
8	0.21124	12.5	7.87	0.0	0.5240	5.631	100.0	6.0821	5.0	311.0	15.2	386.63	29.93
9	0.17004	12.5	7.87	0.0	0.5240	6.004	85.9	6.5921	5.0	311.0	15.2	386.71	17.10

In [8]:

```
df.describe(include='all')
```

Out[8]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	50
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	1

In [9]:

```
df.info()
```

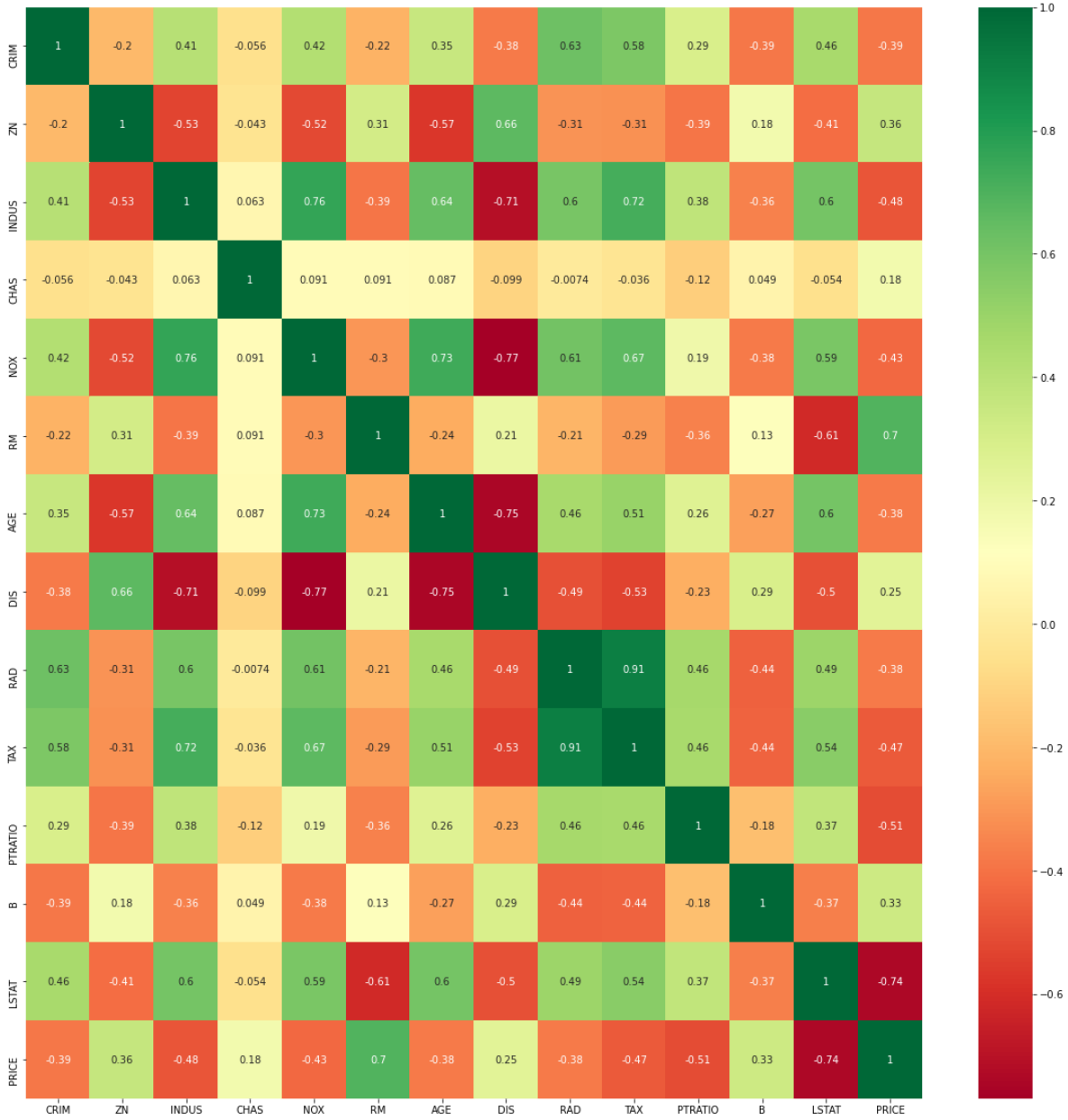
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    float64
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    float64
 9   TAX      506 non-null    float64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    506 non-null    float64
 13  PRICE    506 non-null    float64
dtypes: float64(14)
memory usage: 55.5 KB
```

In [10]:

```
# Correlation

import seaborn as sns
import matplotlib.pyplot as plt

#get correlations of each pair of features in the data
corrmat = df.corr()
top_corr_features = corrmat.index
plt.figure(figsize=(20, 20))

#plot heat map
sns.heatmap(df[top_corr_features].corr(), annot=True, cmap="RdYlGn")
```

Out[10]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x11b830fa0>
```



In [11]:

```
X, y = df.iloc[:, :-1], df.iloc[:, -1]
```

In [12]:

```
print(X.shape)
print(y.shape)
```

```
(506, 13)
(506,)
```

In [13]:

```
from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(X,
                                                                    y,
                                                                    test_size=0.2,
                                                                    shuffle=True,
                                                                    random_state=0)
print(y.shape[0])
print(y_train.shape[0])
print(y_test.shape[0])
```

```
506
404
102
```

In [14]:

```
import xgboost as xgb
xgreg = xgb.XGBRegressor()
xgreg.fit(X_train, y_train)
```

Out[14]:

```
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=6,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=100, n_jobs=0, num_parallel_tree=1, random_state=
0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

In [15]:

```
# r2_score, coefficient of determination
score = xgreg.score(X_train, y_train)
print('r2_score on training data =', score)
```

```
r2_score on training data = 0.9999964964198947
```

In [16]:

```
from sklearn import metrics

print('r2_score on training data =', metrics.r2_score(y_true=y_train,
                                                      y_pred=xgreg.predict(X_train),
                                                      multioutput='variance_weighted'))
```

```
r2_score on training data = 0.9999964964198947
```

In [17]:

```
print('r2_score on test data =', metrics.r2_score(y_true=y_test,
                                                  y_pred=xgreg.predict(X_test),
                                                  multioutput='variance_weighted'))
```

```
r2_score on test data = 0.7375983263596111
```

In [18]:

```
# Hyperparameter Optimization

xgreg = xgb.XGBRegressor()

# Default-Run of default-hyperparameters
parameters = {'learning_rate': [0.3],
              'max_depth': [6],
              'min_child_weight': [1],
              'n_estimators': [100]}


reg_xgb = model_selection.GridSearchCV(estimator=xgreg,
                                       param_grid=parameters,
                                       n_jobs=-1,
                                       cv=3,
                                       refit=True)

reg_xgb.fit(X_train, y_train)
```

Out[18]:

```
GridSearchCV(cv=3,
             estimator=XGBRegressor(base_score=None, booster=None,
                                    colsample_bylevel=None,
                                    colsample_bynode=None,
                                    colsample_bytree=None, gamma=None,
                                    gpu_id=None, importance_type='gain',
                                    interaction_constraints=None,
                                    learning_rate=None, max_delta_step=None,
                                    max_depth=None, min_child_weight=None,
                                    missing=nan, monotone_constraints=None,
                                    n_estimators=100, n_jobs=None,
                                    num_parallel_tree=None, random_state=Non
e,
                                    reg_alpha=None, reg_lambda=None,
                                    scale_pos_weight=None, subsample=None,
                                    tree_method=None, validate_parameters=No
ne,
                                    verbosity=None),
             n_jobs=-1,
             param_grid={'learning_rate': [0.3], 'max_depth': [6],
                         'min_child_weight': [1], 'n_estimators': [100]})
```

In [19]:

```
print(reg_xgb.best_params_)
print(reg_xgb.best_score_)
print(reg_xgb.best_estimator_)
```

```
{'learning_rate': 0.3, 'max_depth': 6, 'min_child_weight': 1, 'n_estimator
s': 100}
0.8265796897665796
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=0.3, max_delta_step=0, max_depth=6,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=100, n_jobs=0, num_parallel_tree=1, random_state=
0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

In [20]:

```
# 1st-Run for best hyperparameters
parameters = {'learning_rate': [0.1, 0.2, 0.3, 0.4, 0.5],
              'max_depth': [2, 4, 6, 8, 10],
              'min_child_weight': [1, 10, 20, 30, 40, 50],
              'n_estimators': [100, 200, 300, 400, 500]}


reg_xgb = model_selection.GridSearchCV(estimator=xgreg,
                                       param_grid=parameters,
                                       n_jobs=-1,
                                       cv=3,
                                       refit=True)

reg_xgb.fit(X_train, y_train)
```

Out[20]:

```
GridSearchCV(cv=3,
             estimator=XGBRegressor(base_score=None, booster=None,
                                    colsample_bylevel=None,
                                    colsample_bynode=None,
                                    colsample_bytree=None, gamma=None,
                                    gpu_id=None, importance_type='gain',
                                    interaction_constraints=None,
                                    learning_rate=None, max_delta_step=None,
                                    max_depth=None, min_child_weight=None,
                                    missing=nan, monotone_constraints=None,
                                    n_estimators=100, n_jobs=None,
                                    num_parallel_tree=None, random_state=Non
e,
                                    reg_alpha=None, reg_lambda=None,
                                    scale_pos_weight=None, subsample=None,
                                    tree_method=None, validate_parameters=No
ne,
                                    verbosity=None),
             n_jobs=-1,
             param_grid={'learning_rate': [0.1, 0.2, 0.3, 0.4, 0.5],
                         'max_depth': [2, 4, 6, 8, 10],
                         'min_child_weight': [1, 10, 20, 30, 40, 50],
                         'n_estimators': [100, 200, 300, 400, 500]})
```

In [21]:

```
print(reg_xgb.best_params_)
print(reg_xgb.best_score_)
print(reg_xgb.best_estimator_)
```

```
{'learning_rate': 0.4, 'max_depth': 4, 'min_child_weight': 20, 'n_estimator
s': 100}
0.875135968698512
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=0.4, max_delta_step=0, max_depth=4,
             min_child_weight=20, missing=nan, monotone_constraints='()',
             n_estimators=100, n_jobs=0, num_parallel_tree=1, random_state=
0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

In [22]:

```
# 2nd-Run for best hyperparameters
parameters = {'learning_rate': [0.3, 0.35, 0.4, 0.45, 0.5],
              'max_depth': [2, 3, 4, 5, 6],
              'min_child_weight': [10, 15, 20, 25, 30],
              'n_estimators': [50, 100, 150, 200, 250]}


reg_xgb = model_selection.GridSearchCV(estimator=xgreg,
                                       param_grid=parameters,
                                       n_jobs=-1,
                                       cv=3,
                                       refit=True)

reg_xgb.fit(X_train, y_train)
```

Out[22]:

```
GridSearchCV(cv=3,
             estimator=XGBRegressor(base_score=None, booster=None,
                                    colsample_bylevel=None,
                                    colsample_bynode=None,
                                    colsample_bytree=None, gamma=None,
                                    gpu_id=None, importance_type='gain',
                                    interaction_constraints=None,
                                    learning_rate=None, max_delta_step=None,
                                    max_depth=None, min_child_weight=None,
                                    missing=nan, monotone_constraints=None,
                                    n_estimators=100, n_jobs=None,
                                    num_parallel_tree=None, random_state=Non
e,
                                    reg_alpha=None, reg_lambda=None,
                                    scale_pos_weight=None, subsample=None,
                                    tree_method=None, validate_parameters=No
ne,
                                    verbosity=None),
             n_jobs=-1,
             param_grid={'learning_rate': [0.3, 0.35, 0.4, 0.45, 0.5],
                         'max_depth': [2, 3, 4, 5, 6],
                         'min_child_weight': [10, 15, 20, 25, 30],
                         'n_estimators': [50, 100, 150, 200, 250]})
```

In [23]:

```
print(reg_xgb.best_params_)
print(reg_xgb.best_score_)
print(reg_xgb.best_estimator_)
```

```
{'learning_rate': 0.4, 'max_depth': 4, 'min_child_weight': 20, 'n_estimator
s': 100}
0.875135968698512
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=0.4, max_delta_step=0, max_depth=4,
             min_child_weight=20, missing=nan, monotone_constraints='()',
             n_estimators=100, n_jobs=0, num_parallel_tree=1, random_state=
0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

In [24]:

```
# 3rd-Run for best hyperparameters
parameters = {'learning_rate': [0.35, 0.375, 0.4, 0.425, 0.45],
              'max_depth': [3, 4, 5, 6, 7],
              'min_child_weight': [15, 18, 20, 22, 25],
              'n_estimators': [80, 90, 100, 110, 120]}


reg_xgb = model_selection.GridSearchCV(estimator=xgreg,
                                       param_grid=parameters,
                                       n_jobs=-1,
                                       cv=3,
                                       refit=True)

reg_xgb.fit(X_train, y_train)
```

Out[24]:

```
GridSearchCV(cv=3,
             estimator=XGBRegressor(base_score=None, booster=None,
                                    colsample_bylevel=None,
                                    colsample_bynode=None,
                                    colsample_bytree=None, gamma=None,
                                    gpu_id=None, importance_type='gain',
                                    interaction_constraints=None,
                                    learning_rate=None, max_delta_step=None,
                                    max_depth=None, min_child_weight=None,
                                    missing=nan, monotone_constraints=None,
                                    n_estimators=100, n_jobs=None,
                                    num_parallel_tree=None, random_state=Non
e,
                                    reg_alpha=None, reg_lambda=None,
                                    scale_pos_weight=None, subsample=None,
                                    tree_method=None, validate_parameters=No
ne,
                                    verbosity=None),
             n_jobs=-1,
             param_grid={'learning_rate': [0.35, 0.375, 0.4, 0.425, 0.45],
                         'max_depth': [3, 4, 5, 6, 7],
                         'min_child_weight': [15, 18, 20, 22, 25],
                         'n_estimators': [80, 90, 100, 110, 120]})
```

In [25]:

```
print(reg_xgb.best_params_)
print(reg_xgb.best_score_)
print(reg_xgb.best_estimator_)
```

```
{'learning_rate': 0.425, 'max_depth': 3, 'min_child_weight': 22, 'n_estimato
rs': 120}
0.8779593850474777
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=0.425, max_delta_step=0, max_depth=3,
             min_child_weight=22, missing=nan, monotone_constraints='()',
             n_estimators=120, n_jobs=0, num_parallel_tree=1, random_state=
0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

In [26]:

```
print('r2_score on test data =', metrics.r2_score(y_true=y_test,
                                                  y_pred=reg_xgb.predict(X_test),
                                                  multioutput='variance_weighted'))
```

```
r2_score on test data = 0.7639250257078705
```

In [27]:

```
final_model = xgb.XGBRegressor(learning_rate=0.425,
                               max_depth=3,
                               min_child_weight=22,
                               n_estimators=120)

final_model.fit(X_train,y_train)
print('r2_score on train data =', metrics.r2_score(y_true=y_train,
                                                   y_pred=reg_xgb.predict(X_train),
                                                   multioutput='variance_weighted'))

print('r2_score on test data =', metrics.r2_score(y_true=y_test,
                                                  y_pred=reg_xgb.predict(X_test),
                                                  multioutput='variance_weighted'))
```

```
r2_score on train data = 0.9922625518853955
r2_score on test data = 0.7639250257078705
```

Kaggle Practice Exercise

This is a practice session in which we will solve the Kaggle competition TalkingData AdTracking Fraud Detection Challenge. The training set, as well as the solution, is provided in here. Please implement the code and try tuning the different parameters to achieve a better result, but first, let's understand why it is beneficial to participate in Kaggle competitions.

Note that this is given to you as an exercise by professor Raghavan where you are requested to read the notebook, modify and improvise the code, and optionally participate in the Kaggle competition. A great way to learn from other people on Kaggle is to read the kernels shared by them and incorporate useful things into your own code.

Please download the following files:

iPython Notebook Download

Training data Download

NOTE: We have used only a fraction of the training set (train_sample, 100k rows), the full training data on Kaggle (train.csv) has about 180 million rows. For better understanding, you are required to implement the same code on a significant portion of the training dataset.

Hope you have gone through the Notebook. Let's answer the following questions:

Question 2/3 Mandatory

Number of Fits

How many fits do we need to do if we define the param_grid as follows in a 5-fold crossvalidation?

```
param_grid = {"base_estimator__max_depth" : [2, 5],
              "learning_rate": [0.2, 0.6, 0.9],
              "subsample": [0.3, 0.6, 0.9]
               }
```

90 ✓ Correct

Feedback:
We have 2*3*3 = 18 combinations of parameter values. On this, performing a 5-fold cross-validation set 18*5 = 90 fits.

18

Your answer is Correct. Attempt 2 of 2 Continue

Additional References

- Installation Guide: XGBoost

Report an error

Please download the following files:

iPython Notebook Download

Training data Download

NOTE: We have used only a fraction of the training set (train_sample, 100k rows), the full training data on Kaggle (train.csv) has about 180 million rows. For better understanding, you are required to implement the same code on a significant portion of the training dataset.

Hope you have gone through the Notebook. Let's answer the following questions:

Question 3/3 Mandatory

Trade Off

We tune learning_rate and n_estimators together.

True ✕ Incorrect

Feedback:

False Correct

Feedback:
There is a trade-off between learning_rate and n_estimators. Hence, we tune only one of them.

Your answer is Wrong. Attempt 1 of 1 Continue

Additional References

- Installation Guide: XGBoost

Report an error

PREVIOUS
XGBoost Lab

NEXT
Summary

TalkingData: Fraudulent Click Prediction

In this notebook, we will apply various boosting algorithms to solve an interesting classification problem from the domain of 'digital fraud'.

The analysis is divided into the following sections:

- Understanding the business problem
- Understanding and exploring the data
- Feature engineering: Creating new features
- Model building and evaluation: AdaBoost
- Modelling building and evaluation: Gradient Boosting
- Modelling building and evaluation: XGBoost

Understanding the Business Problem

TalkingData (https://www.talkingdata.com/) is a Chinese big data company, and one of their areas of expertise is mobile advertisements.

In mobile advertisements, **click fraud** is a major source of losses. Click fraud is the practice of repeatedly clicking on an advertisement hosted on a website with the intention of generating revenue for the host website or draining revenue from the advertiser.

In this case, TalkingData happens to be serving the advertisers (their clients). TalkingData cover a whopping **approx. 70% of the active mobile devices in China**, of which 90% are potentially fraudulent (i.e. the user is actually not going to download the app after clicking).

You can imagine the amount of money they can help clients save if they are able to predict whether a given click is fraudulent (or equivalently, whether a given click will result in a download).

Their current approach to solve this problem is that they've generated a blacklist of IP addresses - those IPs which produce lots of clicks, but never install any apps. Now, they want to try some advanced techniques to predict the probability of a click being genuine/fraud.

In this problem, we will use the features associated with clicks, such as IP address, operating system, device type, time of click etc. to predict the probability of a click being fraud.

They have released the problem on Kaggle here. (https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection).

Understanding and Exploring the Data

The data contains observations of about 240 million clicks, and whether a given click resulted in a download or not (1/0).

On Kaggle, the data is split into train.csv and train_sample.csv (100,000 observations). We'll use the smaller train_sample.csv in this notebook for speed, though while training the model for Kaggle submissions, the full training data will obviously produce better results.

The detailed data dictionary is mentioned here:

- `ip` : ip address of click.
- `app` : app id for marketing.
- `device` : device type id of user mobile phone (e.g., iphone 6 plus, iphone 7, huawei mate 7, etc.)
- `os` : os version id of user mobile phone
- `channel` : channel id of mobile ad publisher
- `click_time` : timestamp of click (UTC)
- `attributed_time` : if user download the app for after clicking an ad, this is the time of the app download
- `is_attributed` : the target that is to be predicted, indicating the app was downloaded

Let's try finding some useful trends in the data.

In [135]:

```
import numpy as np
import pandas as pd
import sklearn
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.cross_validation import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn import metrics

import xgboost as xgb
from xgboost import XGBClassifier
from xgboost import plot_importance
import gc # for deleting unused variables
%matplotlib inline

import os
import warnings
warnings.filterwarnings('ignore')
```

Reading the Data

The code below reads the train_sample.csv file if you set testing = True, else reads the full train.csv file. You can read the sample while tuning the model etc., and then run the model on the full data once done.

Important Note: Save memory when the data is huge

Since the training data is quite huge, the program will be quite slow if you don't consciously follow some best practices to save memory. This notebook demonstrates some of those practices.

In [3]:

```
# reading training data

# specify column dtypes to save memory (by default pandas reads some columns as floats)
dtypes = {
        'ip'            : 'uint16',
        'app'           : 'uint16',
        'device'        : 'uint16',
        'os'            : 'uint16',
        'channel'       : 'uint16',
        'is_attributed' : 'uint8',
        'click_id'      : 'uint32' # note that click_id is only in test data, not training
        }

# read training_sample.csv for quick testing/debug, else read the full train.csv
testing = True
if testing:
    train_path = "train_sample.csv"
    skiprows = None
    nrows = None
    colnames=['ip','app','device','os', 'channel', 'click_time', 'is_attributed']
else:
    train_path = "train.csv"
    skiprows = range(1, 144903891)
    nrows = 10000000
    colnames=['ip','app','device','os', 'channel', 'click_time', 'is_attributed']

# read training data
train_sample = pd.read_csv(train_path, skiprows=skiprows, nrows=nrows, dtype=dtypes, usecol
```

In [4]:

```
# length of training data
len(train_sample.index)
```

Out[4]:

```
100000
```

In [9]:

```
# Displays memory consumed by each column ---
print(train_sample.memory_usage())
```

```
Index              80
ip             200000
app            200000
device         200000
os             200000
channel        200000
click_time     800000
is_attributed  100000
dtype: int64
```

In [6]:

```
# space used by training data
print('Training dataset uses {0} MB'.format(train_sample.memory_usage().sum()/1024**2))
```

```
Training dataset uses 1.8120574951171875 MB
```

In [8]:

```
# training data top rows
train_sample.head()
```

Out[8]:

	ip	app	device	os	channel	click_time	is_attributed
0	22004	12	1	13	497	2017-11-07 09:30:38	0
1	40024	25	1	17	259	2017-11-07 13:40:27	0
2	35888	12	1	19	212	2017-11-07 18:05:24	0
3	29048	13	1	13	477	2017-11-07 04:58:08	0
4	2877	12	1	1	178	2017-11-09 09:00:09	0

Exploring the Data - Univariate Analysis

Let's now understand and explore the data. Let's start with understanding the size and data types of the train_sample data.

In [11]:

```
# look at non-null values, number of entries etc.
# there are no missing values
train_sample.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 7 columns):
ip              100000 non-null uint16
app             100000 non-null uint16
device          100000 non-null uint16
os              100000 non-null uint16
channel         100000 non-null uint16
click_time      100000 non-null object
is_attributed   100000 non-null uint8
dtypes: object(1), uint16(5), uint8(1)
memory usage: 1.8+ MB
```

In [15]:

```
# Basic exploratory analysis

# Number of unique values in each column
def fraction_unique(x):
    return len(train_sample[x].unique())

number_unique_vals = {x: fraction_unique(x) for x in train_sample.columns}
number_unique_vals
```

Out[15]:

```
{'app': 161,
 'channel': 161,
 'click_time': 80350,
 'device': 100,
 'ip': 28470,
 'is_attributed': 2,
 'os': 130}
```

In [23]:

```
# All columns apart from click time are originally int type,
# though note that they are all actually categorical
train_sample.dtypes
```

Out[23]:

```
ip               uint16
app              uint16
device           uint16
os               uint16
channel          uint16
click_time       object
is_attributed     uint8
dtype: object
```

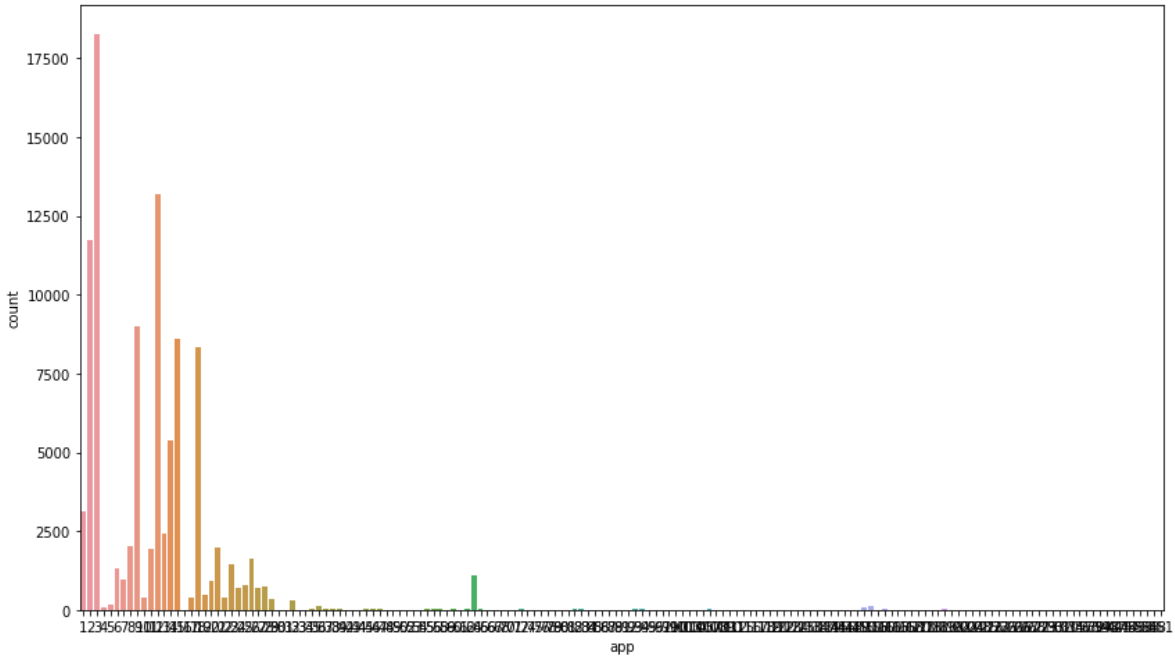
There are certain 'apps' which have quite high number of instances/rows (each row is a click). The plot below shows this.

In [24]:

```
# # distribution of 'app'
# # some 'apps' have a disproportionately high number of clicks (>15k), and some are very r
plt.figure(figsize=(14, 8))
sns.countplot(x="app", data=train_sample)
```

Out[24]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x10beb5518>
```

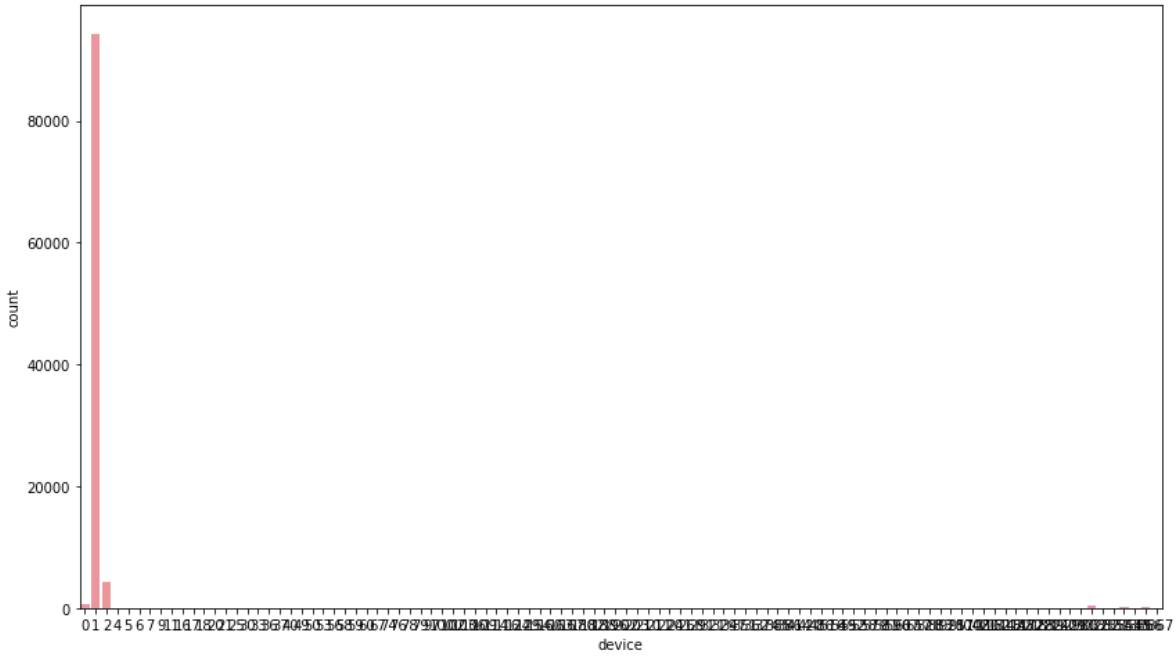


In [25]:

```
# # distribution of 'device'
# # this is expected because a few popular devices are used heavily
plt.figure(figsize=(14, 8))
sns.countplot(x="device", data=train_sample)
```

Out[25]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x10bfd9978>
```

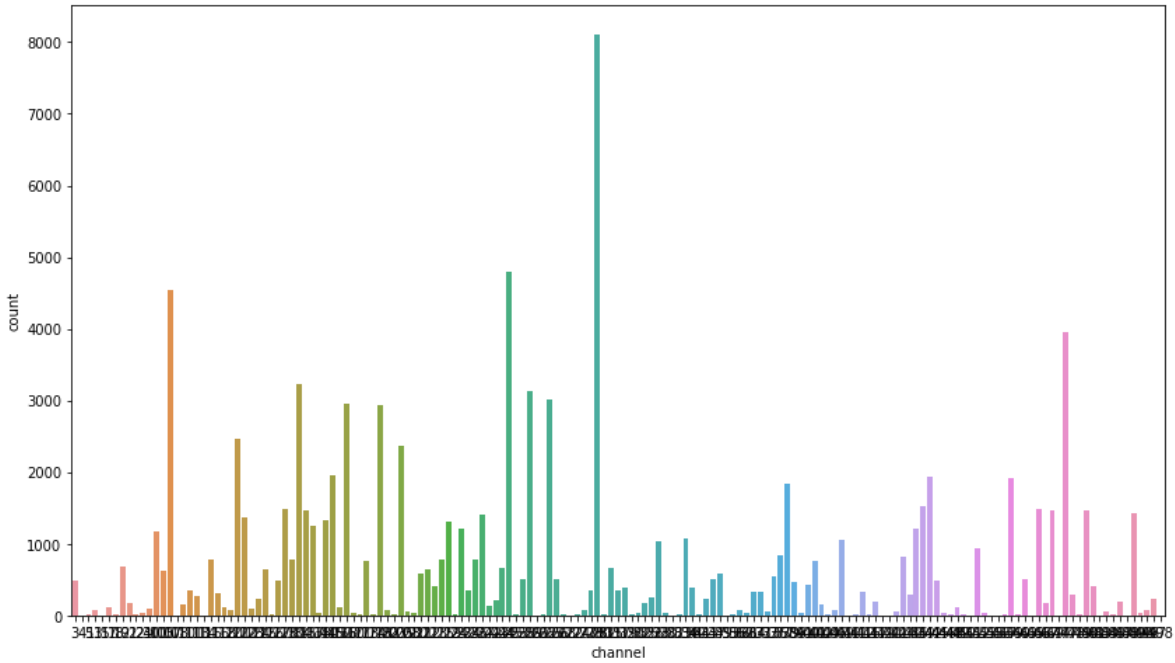


In [26]:

```
# # channel: various channels get clicks in comparable quantities
plt.figure(figsize=(14, 8))
sns.countplot(x="channel", data=train_sample)
```

Out[26]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x10c523940>
```

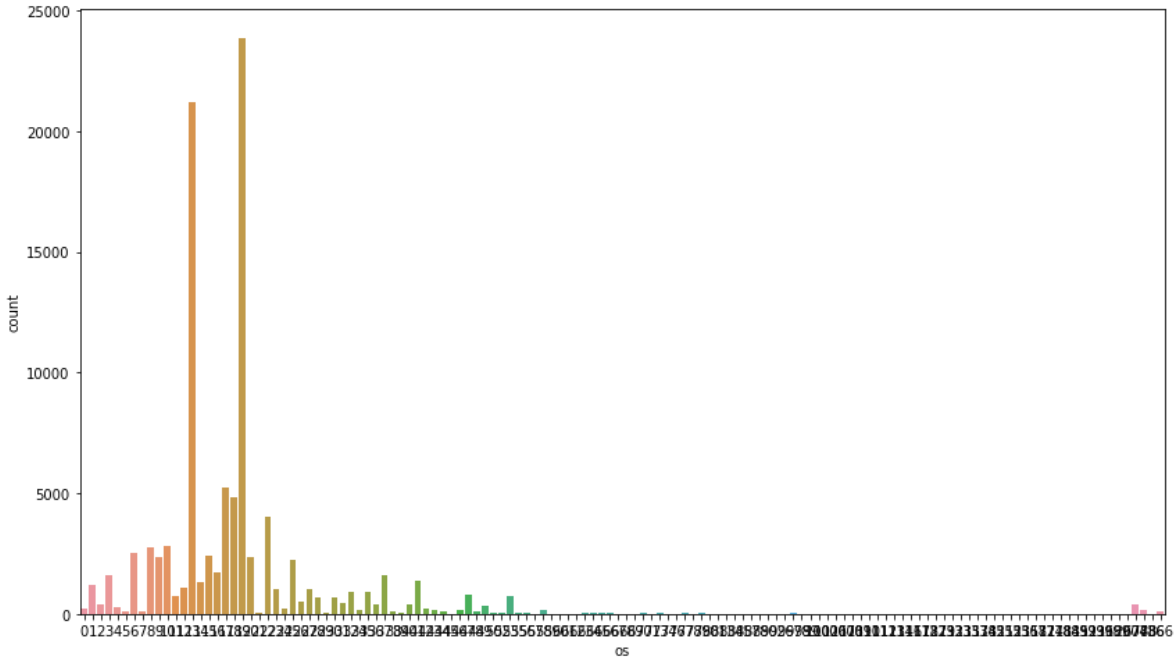


In [27]:

```
# # os: there are a couple commos OSes (android and ios?), though some are rare and can ind
plt.figure(figsize=(14, 8))
sns.countplot(x="os", data=train_sample)
```

Out[27]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x10ba40ef0>
```



Let's now look at the distribution of the target variable 'is_attributed'.

In [28]:

```
# # target variable distribution
100*(train_sample['is_attributed'].astype('object').value_counts()/len(train_sample.index))
```

Out[28]:

```
0    99.773
1     0.227
Name: is_attributed, dtype: float64
```

Only **about 0.2% of clicks are 'fraudulent'**, which is expected in a fraud detection problem. Such high class imbalance is probably going to be the toughest challenge of this problem.

Exploring the Data - Segmented Univariate Analysis

Let's now look at how the target variable varies with the various predictors.

In [35]:

```
# plot the average of 'is_attributed', or 'download rate'
# with app (clearly this is non-readable)
app_target = train_sample.groupby('app').is_attributed.agg(['mean', 'count'])
app_target
```

Out[35]:

app	mean	count
1	0.000000	3135
2	0.000000	11737
3	0.000219	18279
4	0.000000	58
5	0.074468	188
6	0.000000	1303
7	0.000000	981
8	0.001996	2004
9	0.000890	8992
10	0.046392	388

This is clearly non-readable, so let's first get rid of all the apps that are very rare (say which comprise of less than 20% clicks) and plot the rest.

In [63]:

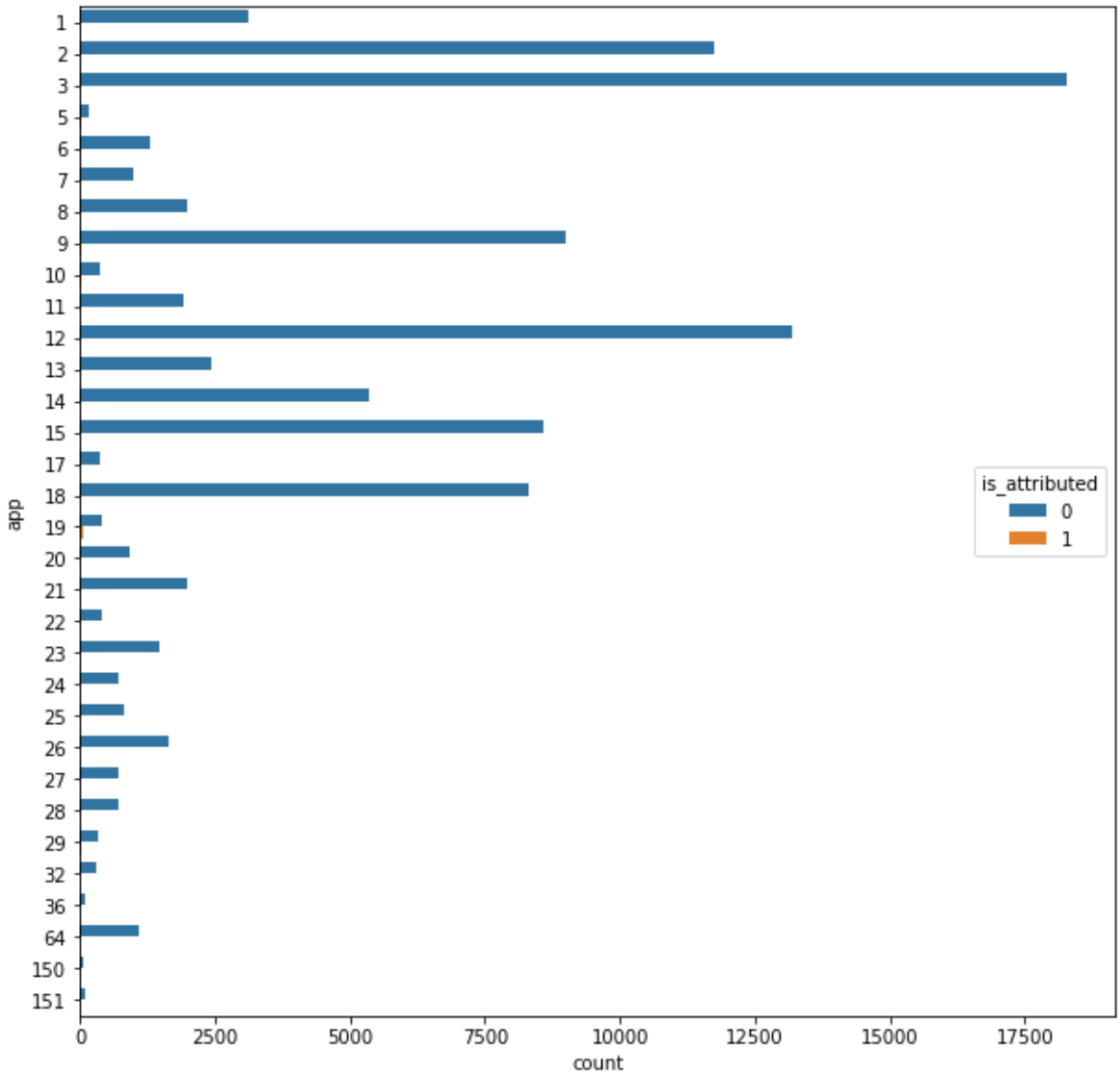
```
frequent_apps = train_sample.groupby('app').size().reset_index(name='count')
frequent_apps = frequent_apps[frequent_apps['count']>frequent_apps['count'].quantile(0.80)]
frequent_apps = frequent_apps.merge(train_sample, on='app', how='inner')
frequent_apps.head()
```

Out[63]:

	app	count	ip	device	os	channel	is_attributed	day_of_week	day_of_year	month	h
0	1	3135	17059	1	17	135	0	3	313	11	
1	1	3135	52432	1	13	115	0	1	311	11	
2	1	3135	23706	1	27	124	0	1	311	11	
3	1	3135	58458	1	19	101	0	3	313	11	
4	1	3135	34067	1	15	134	0	1	311	11	

In [64]:

```
plt.figure(figsize=(10,10))
sns.countplot(y="app", hue="is_attributed", data=frequent_apps);
```



You can do lots of other interesting ananlysis with the existing features. For now, let's create some new features which will probably improve the model.

Feature Engineering

Let's now derive some new features from the existing ones. There are a number of features one can extract from `click_time` itself, and by grouping combinations of IP with other features.

Datetime Based Features

In [45]:

```
# Creating datetime variables
# takes in a df, adds date/time based columns to it, and returns the modified df
def timeFeatures(df):
    # Derive new features using the click_time column
    df['datetime'] = pd.to_datetime(df['click_time'])
    df['day_of_week'] = df['datetime'].dt.dayofweek
    df["day_of_year"] = df["datetime"].dt.dayofyear
    df["month"] = df["datetime"].dt.month
    df["hour"] = df["datetime"].dt.hour
    return df
```

In [46]:

```
# creating new datetime variables and dropping the old ones
train_sample = timeFeatures(train_sample)
train_sample.drop(['click_time', 'datetime'], axis=1, inplace=True)
train_sample.head()
```

Out[46]:

	ip	app	device	os	channel	is_attributed	day_of_week	day_of_year	month	hour
0	22004	12	1	13	497	0	1	311	11	9
1	40024	25	1	17	259	0	1	311	11	13
2	35888	12	1	19	212	0	1	311	11	18
3	29048	13	1	13	477	0	1	311	11	4
4	2877	12	1	1	178	0	3	313	11	9

In [65]:

```
# datatypes
# note that by default the new datetime variables are int64
train_sample.dtypes
```

Out[65]:

```
ip               uint16
app              uint16
device           uint16
os               uint16
channel          uint16
is_attributed     uint8
day_of_week       int64
day_of_year       int64
month             int64
hour              int64
dtype: object
```

In [66]:

```
# memory used by training data
print('Training dataset uses {0} MB'.format(train_sample.memory_usage().sum()/1024**2))
```

```
Training dataset uses 4.1008758544921875 MB
```

In [67]:

```
# lets convert the variables back to lower dtype again
int_vars = ['app', 'device', 'os', 'channel', 'day_of_week','day_of_year', 'month', 'hour']
train_sample[int_vars] = train_sample[int_vars].astype('uint16')
```

In [68]:

```
train_sample.dtypes
```

Out[68]:

```
ip               uint16
app              uint16
device           uint16
os               uint16
channel          uint16
is_attributed     uint8
day_of_week      uint16
day_of_year      uint16
month            uint16
hour             uint16
dtype: object
```

In [69]:

```
# space used by training data
print('Training dataset uses {0} MB'.format(train_sample.memory_usage().sum()/1024**2))
```

```
Training dataset uses 1.8120574951171875 MB
```

IP Grouping Based Features

Let's now create some important features by grouping IP addresses with features such as os, channel, hour, day etc. Also, count of each IP address will also be a feature.

Note that though we are deriving new features by grouping IP addresses, using IP adress itself as a features is not a good idea. This is because (in the test data) if a new IP address is seen, the model will see a new 'category' and will not be able to make predictions (IP is a categorical variable, it has just been encoded with numbers).

In [71]:

```
# number of clicks by count of IP address
# note that we are explicitly asking pandas to re-encode the aggregated features
# as 'int16' to save memory
ip_count = train_sample.groupby('ip').size().reset_index(name='ip_count').astype('int16')
ip_count.head()
```

Out[71]:

	ip	ip_count
0	8	1
1	9	1
2	10	3
3	14	1
4	16	6

We can now merge this dataframe with the original training df. Similarly, we can create combinations of various features such as ip_day_hour (count of ip-day-hour combinations), ip_hour_channel, ip_hour_app, etc.

The following function takes in a dataframe and creates these features.

In [73]:

```
# creates groupings of IP addresses with other features and appends the new features to the
def grouped_features(df):
    # ip_count
    ip_count = df.groupby('ip').size().reset_index(name='ip_count').astype('uint16')
    ip_day_hour = df.groupby(['ip', 'day_of_week', 'hour']).size().reset_index(name='ip_day
    ip_hour_channel = df[['ip', 'hour', 'channel']].groupby(['ip', 'hour', 'channel']).size
    ip_hour_os = df.groupby(['ip', 'hour', 'os']).channel.count().reset_index(name='ip_hour
    ip_hour_app = df.groupby(['ip', 'hour', 'app']).channel.count().reset_index(name='ip_ho
    ip_hour_device = df.groupby(['ip', 'hour', 'device']).channel.count().reset_index(name=

    # merge the new aggregated features with the df
    df = pd.merge(df, ip_count, on='ip', how='left')
    del ip_count
    df = pd.merge(df, ip_day_hour, on=['ip', 'day_of_week', 'hour'], how='left')
    del ip_day_hour
    df = pd.merge(df, ip_hour_channel, on=['ip', 'hour', 'channel'], how='left')
    del ip_hour_channel
    df = pd.merge(df, ip_hour_os, on=['ip', 'hour', 'os'], how='left')
    del ip_hour_os
    df = pd.merge(df, ip_hour_app, on=['ip', 'hour', 'app'], how='left')
    del ip_hour_app
    df = pd.merge(df, ip_hour_device, on=['ip', 'hour', 'device'], how='left')
    del ip_hour_device

    return df
```

In [75]:

```
train_sample = grouped_features(train_sample)
```

In [76]:

```
train_sample.head()
```

Out[76]:

	ip	app	device	os	channel	is_attributed	day_of_week	day_of_year	month	hour	ip_
0	22004	12	1	13	497	0	1	311	11	9	
1	40024	25	1	17	259	0	1	311	11	13	
2	35888	12	1	19	212	0	1	311	11	18	
3	29048	13	1	13	477	0	1	311	11	4	
4	2877	12	1	1	178	0	3	313	11	9	

In [77]:

```
print('Training dataset uses {0} MB'.format(train_sample.memory_usage().sum()/1024**2))
```

```
Training dataset uses 3.719329833984375 MB
```

In [79]:

```
# garbage collect (unused) object
gc.collect()
```

Out[79]:

```
17431
```

Modelling

Let's now build models to predict the variable `is_attributed` (downloaded). We'll try the several variants of boosting (adaboost, gradient boosting and XGBoost), tune the hyperparameters in each model and choose the one which gives the best performance.

In the original Kaggle competition, the metric for model evaluation is **area under the ROC curve**.

In [83]:

```
# create x and y train
X = train_sample.drop('is_attributed', axis=1)
y = train_sample[['is_attributed']]

# split data into train and test/validation sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=101)
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(80000, 15)
(80000, 1)
(20000, 15)
(20000, 1)
```

In [84]:

```
# check the average download rates in train and test data, should be comparable
print(y_train.mean())
print(y_test.mean())
```

```
is_attributed    0.002275
dtype: float64
is_attributed    0.00225
dtype: float64
```

AdaBoost

In [105]:

```
# adaboost classifier with max 600 decision trees of depth=2
# learning_rate/shrinkage=1.5

# base estimator
tree = DecisionTreeClassifier(max_depth=2)

# adaboost with the tree as base estimator
adaboost_model_1 = AdaBoostClassifier(
    base_estimator=tree,
    n_estimators=600,
    learning_rate=1.5,
    algorithm="SAMME")
```

In [106]:

```
# fit
adaboost_model_1.fit(X_train, y_train)
```

Out[106]:

```
AdaBoostClassifier(algorithm='SAMME',
          base_estimator=DecisionTreeClassifier(class_weight=None, criterion
='gini', max_depth=2,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
          learning_rate=1.5, n_estimators=600, random_state=None)
```

In [107]:

```
# predictions
# the second column represents the probability of a click resulting in a download
predictions = adaboost_model_1.predict_proba(X_test)
predictions[:10]
```

Out[107]:

```
array([[ 0.5259697 ,  0.4740303 ],
       [ 0.52720083,  0.47279917],
       [ 0.533081  ,  0.466919  ],
       [ 0.52194781,  0.47805219],
       [ 0.51032691,  0.48967309],
       [ 0.52721323,  0.47278677],
       [ 0.5183883 ,  0.4816117 ],
       [ 0.52170927,  0.47829073],
       [ 0.52412251,  0.47587749],
       [ 0.51552875,  0.48447125]])
```

In [108]:

```
# metrics: AUC
metrics.roc_auc_score(y_test, predictions[:,1])
```

Out[108]:

```
0.92838553411843305
```

AdaBoost - Hyperparameter Tuning

Let's now tune the hyperparameters of the AdaBoost classifier. In this case, we have two types of hyperparameters - those of the component trees (max_depth etc.) and those of the ensemble (n_estimators, learning_rate etc.).

We can tune both using the following technique - the keys of the form `base_estimator_parameter_name` belong to the trees (base estimator), and the rest belong to the ensemble.

In [171]:

```
# parameter grid
param_grid = {"base_estimator__max_depth" : [2, 5],
              "n_estimators": [200, 400, 600]
             }
```

In [163]:

```
# base estimator
tree = DecisionTreeClassifier()

# adaboost with the tree as base estimator
# learning rate is arbitrarily set to 0.6, we'll discuss learning_rate below
ABC = AdaBoostClassifier(
    base_estimator=tree,
    learning_rate=0.6,
    algorithm="SAMME")
```

In [164]:

```
# run grid search
folds = 3
grid_search_ABC = GridSearchCV(ABC,
                               cv = folds,
                               param_grid=param_grid,
                               scoring = 'roc_auc',
                               return_train_score=True,
                               verbose = 1)
```

In [165]:

```
# fit
grid_search_ABC.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits

[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed: 10.1min finished
```

Out[165]:

```
GridSearchCV(cv=3, error_score='raise',
       estimator=AdaBoostClassifier(algorithm='SAMME',
          base_estimator=DecisionTreeClassifier(class_weight=None, criterion
='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
          learning_rate=0.6, n_estimators=50, random_state=None),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'n_estimators': [200, 400, 600], 'base_estimator__max_dep
th': [2, 5]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring='roc_auc', verbose=1)
```

In [166]:

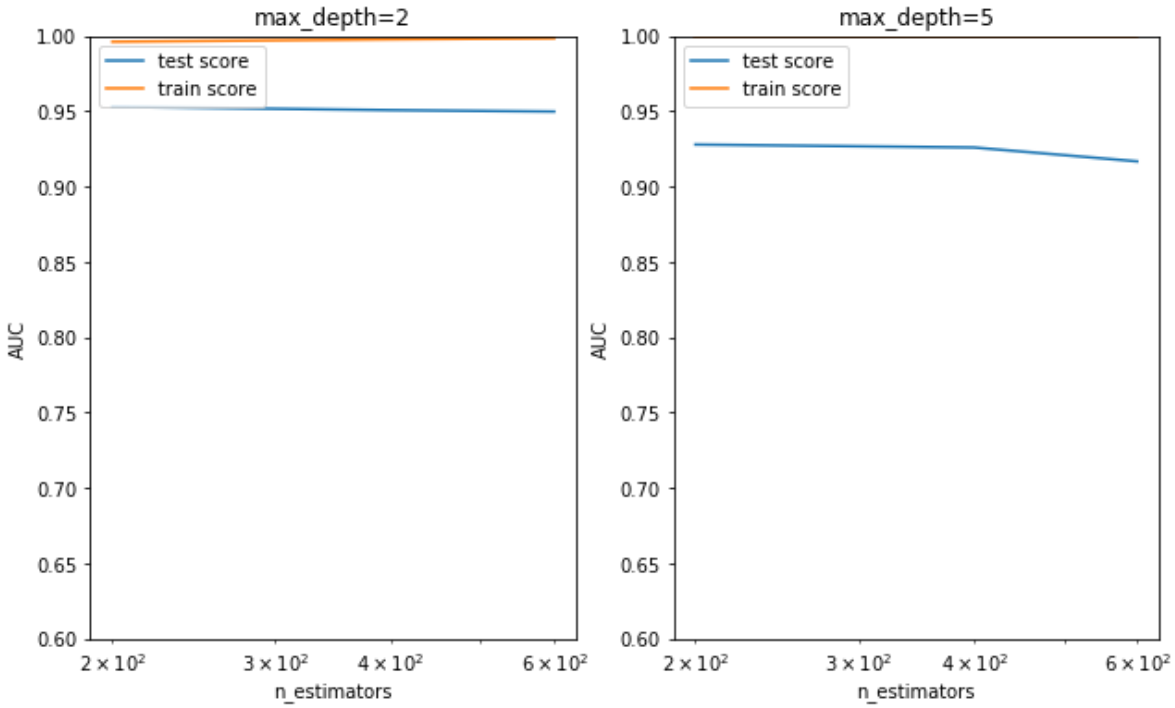
```
# cv results
cv_results = pd.DataFrame(grid_search_ABC.cv_results_)
cv_results
```

Out[166]:

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_base_estimator_
0	9.376183	0.209405	0.952831	0.995954	
1	19.704252	0.450558	0.950575	0.997556	
2	30.060021	0.651644	0.949670	0.998278	
3	22.058344	0.310703	0.927757	1.000000	
4	44.099881	0.596813	0.925812	1.000000	
5	67.404276	0.879526	0.916619	1.000000	

In [170]:

```
# plotting AUC with hyperparameter combinations

plt.figure(figsize=(16,6))
for n, depth in enumerate(param_grid['base_estimator__max_depth']):


    # subplot 1/n
    plt.subplot(1,3, n+1)
    depth_df = cv_results[cv_results['param_base_estimator__max_depth']==depth]

    plt.plot(depth_df["param_n_estimators"], depth_df["mean_test_score"])
    plt.plot(depth_df["param_n_estimators"], depth_df["mean_train_score"])
    plt.xlabel('n_estimators')
    plt.ylabel('AUC')
    plt.title("max_depth={0}".format(depth))
    plt.ylim([0.60, 1])
    plt.legend(['test score', 'train score'], loc='upper left')
    plt.xscale('log')
```



The results above show that:

- The ensemble with max_depth=5 is clearly overfitting (training auc is almost 1, while the test score is much lower)
- At max_depth=2, the model performs slightly better (approx 95% AUC) with a higher test score

Thus, we should go ahead with `max_depth=2` and `n_estimators=200` .

Note that we haven't experimented with many other important hyperparameters till now, such as `learning rate` , `subsample` etc., and the results might be considerably improved by tuning them. We'll next experiment with these hyperparameters.

In [183]:

```
# model performance on test data with chosen hyperparameters

# base estimator
tree = DecisionTreeClassifier(max_depth=2)

# adaboost with the tree as base estimator
# learning rate is arbitrarily set, we'll discuss learning_rate below
ABC = AdaBoostClassifier(
    base_estimator=tree,
    learning_rate=0.6,
    n_estimators=200,
    algorithm="SAMME")

ABC.fit(X_train, y_train)
```

Out[183]:

```
AdaBoostClassifier(algorithm='SAMME',
          base_estimator=DecisionTreeClassifier(class_weight=None, criterion
='gini', max_depth=2,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
          learning_rate=0.6, n_estimators=200, random_state=None)
```

In [184]:

```
# predict on test data
predictions = ABC.predict_proba(X_test)
predictions[:10]
```

Out[184]:

```
array([[ 0.5880972 ,  0.4119028 ],
       [ 0.58960261,  0.41039739],
       [ 0.60708804,  0.39291196],
       [ 0.57134614,  0.42865386],
       [ 0.55591021,  0.44408979],
       [ 0.58624788,  0.41375212],
       [ 0.56320517,  0.43679483],
       [ 0.58981139,  0.41018861],
       [ 0.59090843,  0.40909157],
       [ 0.56433022,  0.43566978]])
```

In [185]:

```
# roc auc
metrics.roc_auc_score(y_test, predictions[:, 1])
```

Out[185]:

```
0.94789331551546541
```

Gradient Boosting Classifier

Let's now try the gradient boosting classifier. We'll experiment with two main hyperparameters now - `learning_rate` (shrinkage) and `subsample`.

By adjusting the learning rate to less than 1, we can regularize the model. A model with higher learning_rate learns fast, but is prone to overfitting; one with a lower learning rate learns slowly, but avoids overfitting.

Also, there's a trade-off between `learning_rate` and `n_estimators` - the higher the learning rate, the lesser trees the model needs (and thus we usually tune only one of them).

Also, by subsampling (setting `subsample` to less than 1), we can have the individual models built on random subsamples of size `subsample`. That way, each tree will be trained on different subsets and reduce the model's variance.

In [186]:

```
# parameter grid
param_grid = {"learning_rate": [0.2, 0.6, 0.9],
              "subsample": [0.3, 0.6, 0.9]
             }
```

In [187]:

```
# adaboost with the tree as base estimator
GBC = GradientBoostingClassifier(max_depth=2, n_estimators=200)
```

In [155]:

```
# run grid search
folds = 3
grid_search_GBC = GridSearchCV(GBC,
                               cv = folds,
                               param_grid=param_grid,
                               scoring = 'roc_auc',
                               return_train_score=True,
                               verbose = 1)

grid_search_GBC.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 9 candidates, totalling 27 fits

[Parallel(n_jobs=1)]: Done  27 out of  27 | elapsed:  6.8min finished
```

Out[155]:

```
GridSearchCV(cv=3, error_score='raise',
       estimator=GradientBoostingClassifier(criterion='friedman_mse', init=N
one,
              learning_rate=0.1, loss='deviance', max_depth=400,
              max_features=None, max_leaf_nodes=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=100,
              presort='auto', random_state=None, subsample=1.0, verbose=0,
              warm_start=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'learning_rate': [0.2, 0.6, 0.9], 'subsample': [0.3, 0.6,
0.9]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring='roc_auc', verbose=1)
```

In [199]:

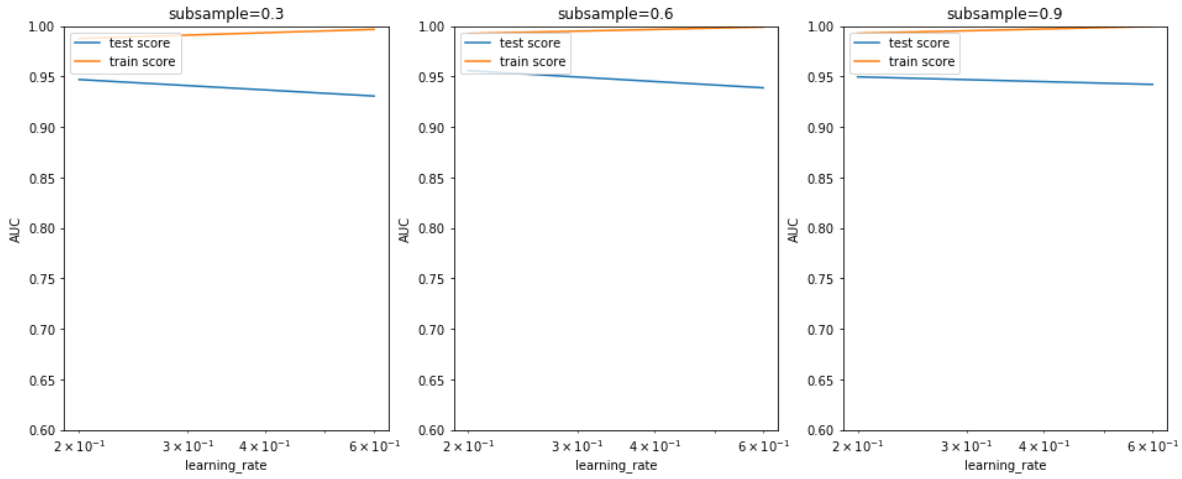
```
cv_results = pd.DataFrame(grid_search_GBC.cv_results_)
cv_results.head()
```

Out[199]:

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_learning_rate	param_subsam
0	16.663277	0.110181	0.928349	0.997548	0.2	
1	19.186795	0.112623	0.746107	0.999029	0.2	
2	9.152588	0.058281	0.567046	0.999806	0.2	
3	16.168680	0.110494	0.897310	0.997573	0.6	

In [207]:

```
# # plotting
plt.figure(figsize=(16,6))


for n, subsample in enumerate(param_grid['subsample']):


    # subplot 1/n
    plt.subplot(1,len(param_grid['subsample']), n+1)
    df = cv_results[cv_results['param_subsample']==subsample]

    plt.plot(df["param_learning_rate"], df["mean_test_score"])
    plt.plot(df["param_learning_rate"], df["mean_train_score"])
    plt.xlabel('learning_rate')
    plt.ylabel('AUC')
    plt.title("subsample={0}".format(subsample))
    plt.ylim([0.60, 1])
    plt.legend(['test score', 'train score'], loc='upper left')
    plt.xscale('log')
```



It is clear from the plot above that the model with a lower subsample ratio performs better, while those with higher subsamples tend to overfit.

Also, a lower learning rate results in less overfitting.

XGBoost

Let's finally try XGBoost. The hyperparameters are the same, some important ones being `subsample`, `learning_rate`, `max_depth` etc.

In [188]:

```
# fit model on training data with default hyperparameters
model = XGBClassifier()
model.fit(X_train, y_train)
```

Out[188]:

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
       n_jobs=1, nthread=None, objective='binary:logistic', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1)
```

In [189]:

```
# make predictions for test data
# use predict_proba since we need probabilities to compute auc
y_pred = model.predict_proba(X_test)
y_pred[:10]
```

Out[189]:

```
array([[  9.99876201e-01,   1.23777572e-04],
       [  9.99802351e-01,   1.97641290e-04],
       [  9.99812186e-01,   1.87795449e-04],
       [  9.99353409e-01,   6.46599103e-04],
       [  9.98394549e-01,   1.60545984e-03],
       [  9.99828875e-01,   1.71130727e-04],
       [  9.99547005e-01,   4.52976237e-04],
       [  9.99449134e-01,   5.50867291e-04],
       [  9.99769509e-01,   2.30486505e-04],
       [  9.96964693e-01,   3.03530321e-03]], dtype=float32)
```

In [190]:

```
# evaluate predictions
roc = metrics.roc_auc_score(y_test, y_pred[:, 1])
print("AUC: %.2f%%" % (roc * 100.0))
```

```
AUC: 94.85%
```

The roc_auc in this case is about 0.95% with default hyperparameters. Let's try changing the hyperparameters - an exhaustive list of XGBoost hyperparameters is here: http://xgboost.readthedocs.io/en/latest/parameter.html (http://xgboost.readthedocs.io/en/latest/parameter.html)

Let's now try tuning the hyperparameters using k-fold CV. We'll then use grid search CV to find the optimal values of hyperparameters.

In [197]:

```
# hyperparameter tuning with XGBoost

# creating a KFold object
folds = 3

# specify range of hyperparameters
param_grid = {'learning_rate': [0.2, 0.6],
              'subsample': [0.3, 0.6, 0.9]}


# specify model
xgb_model = XGBClassifier(max_depth=2, n_estimators=200)

# set up GridSearchCV()
model_cv = GridSearchCV(estimator = xgb_model,
                        param_grid = param_grid,
                        scoring= 'roc_auc',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)
```

In [198]:

```
# fit the model
model_cv.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits

[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed:  1.3min finished
```

Out[198]:

```
GridSearchCV(cv=3, error_score='raise',
       estimator=XGBClassifier(base_score=0.5, booster='gbtree', colsample_b
ylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=2, min_child_weight=1, missing=None, n_estimators=200,
       n_jobs=1, nthread=None, objective='binary:logistic', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'learning_rate': [0.2, 0.6], 'subsample': [0.3, 0.6, 0.
9]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring='roc_auc', verbose=1)
```

In [202]:

```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[202]:

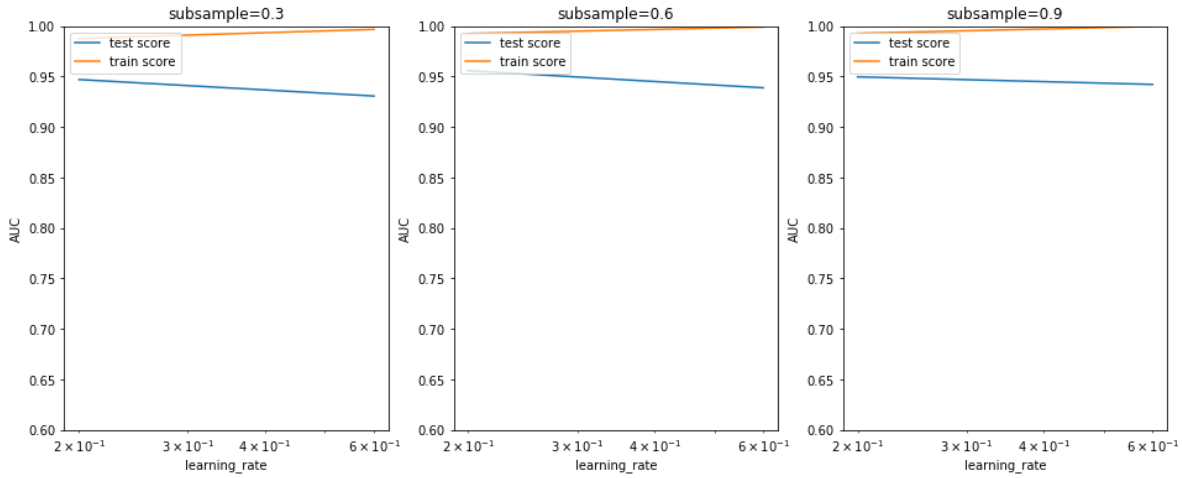
	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_learning_rate	param_subsam
0	3.741100	0.152881	0.946876	0.987553	0.2	
1	4.502307	0.145803	0.955664	0.992432	0.2	
2	3.939432	0.142248	0.949553	0.992693	0.2	
3	3.833046	0.139765	0.930576	0.996570	0.6	

In [ ]:

```
# convert parameters to int for plotting on x-axis
cv_results['param_learning_rate'] = cv_results['param_learning_rate'].astype('float')
cv_results['param_max_depth'] = cv_results['param_max_depth'].astype('float')
cv_results.head()
```

In [203]:

```
# # plotting
plt.figure(figsize=(16,6))

param_grid = {'learning_rate': [0.2, 0.6],
              'subsample': [0.3, 0.6, 0.9]}


for n, subsample in enumerate(param_grid['subsample']):


    # subplot 1/n
    plt.subplot(1,len(param_grid['subsample']), n+1)
    df = cv_results[cv_results['param_subsample']==subsample]

    plt.plot(df["param_learning_rate"], df["mean_test_score"])
    plt.plot(df["param_learning_rate"], df["mean_train_score"])
    plt.xlabel('learning_rate')
    plt.ylabel('AUC')
    plt.title("subsample={0}".format(subsample))
    plt.ylim([0.60, 1])
    plt.legend(['test score', 'train score'], loc='upper left')
    plt.xscale('log')
```



The results show that a subsample size of 0.6 and learning_rate of about 0.2 seems optimal. Also, XGBoost has resulted in the highest ROC AUC obtained (across various hyperparameters).

Let's build a final model with the chosen hyperparameters.

In [204]:

```
# chosen hyperparameters
# 'objective':'binary:logistic' outputs probability rather than label, which we need for au
params = {'learning_rate': 0.2,
          'max_depth': 2,
          'n_estimators':200,
          'subsample':0.6,
         'objective':'binary:logistic'}

# fit model on training data
model = XGBClassifier(params = params)
model.fit(X_train, y_train)
```

Out[204]:

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
       n_jobs=1, nthread=None, objective='binary:logistic',
       params={'n_estimators': 200, 'max_depth': 2, 'learning_rate': 0.2, 's
ubsample': 0.6, 'objective': 'binary:logistic'},
       random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
       seed=None, silent=True, subsample=1)
```

In [205]:

```
# predict
y_pred = model.predict_proba(X_test)
y_pred[:10]
```

Out[205]:

```
array([[  9.99876201e-01,   1.23777572e-04],
       [  9.99802351e-01,   1.97641290e-04],
       [  9.99812186e-01,   1.87795449e-04],
       [  9.99353409e-01,   6.46599103e-04],
       [  9.98394549e-01,   1.60545984e-03],
       [  9.99828875e-01,   1.71130727e-04],
       [  9.99547005e-01,   4.52976237e-04],
       [  9.99449134e-01,   5.50867291e-04],
       [  9.99769509e-01,   2.30486505e-04],
       [  9.96964693e-01,   3.03530321e-03]], dtype=float32)
```

The first column in y_pred is the P(0), i.e. P(not fraud), and the second column is P(1/fraud).

In [206]:

```
# roc_auc
auc = sklearn.metrics.roc_auc_score(y_test, y_pred[:, 1])
auc
```

Out[206]:

```
0.94848687324257352
```

Finally, let's also look at the feature importances.

In [225]:

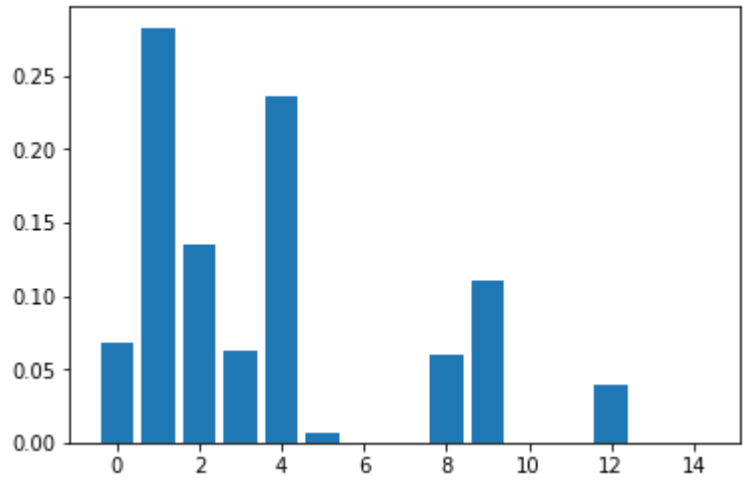
```
# feature importance
importance = dict(zip(X_train.columns, model.feature_importances_))
importance
```

Out[225]:

```
{'app': 0.28301886,
 'channel': 0.23584905,
 'day_of_week': 0.006289308,
 'day_of_year': 0.0,
 'device': 0.13522013,
 'hour': 0.059748426,
 'ip': 0.067610063,
 'ip_count': 0.11006289,
 'ip_day_hour': 0.0,
 'ip_hour_app': 0.0,
 'ip_hour_channel': 0.0,
 'ip_hour_device': 0.0,
 'ip_hour_os': 0.039308175,
 'month': 0.0,
 'os': 0.062893085}
```

In [228]:

```
# plot
plt.bar(range(len(model.feature_importances_)), model.feature_importances_)
plt.show()
```



Predictions on Test Data

Since this problem is hosted on Kaggle, you can choose to make predictions on the test data and submit your results. Please note the following points and recommendations if you go ahead with Kaggle:

Recommendations for training:

- We have used only a fraction of the training set (train_sample, 100k rows), the full training data on Kaggle (train.csv) has about 180 million rows. You'll get good results only if you train the model on a significant portion of the training dataset.
- Because of the size, you'll need to use Kaggle kernels to train the model on full training data. Kaggle kernels provide powerful computation capacities on cloud (for free).

- Even on the kernel, you may need to use a portion of the training dataset (try using the last 20-30 million rows).
- Make sure you save memory by following some tricks and best practices, else you won't be able to train the model at all on a large dataset.

In [ ]:

```
# # read submission file
# sample_sub = pd.read_csv(path+'sample_submission.csv')
# sample_sub.head()
```

In [ ]:

```
# # predict probability of test data
# test_final = pd.read_csv(path+'test.csv')
# test_final.head()
```

In [ ]:

```
# # predictions on test data
# test_final = timeFeatures(test_final)
# test_final.head()
```

In [ ]:

```
# test_final.drop(['click_time', 'datetime'], axis=1, inplace=True)
```

In [ ]:

```
# test_final.head()
```

In [ ]:

```
# test_final[categorical_cols]=test_final[categorical_cols].apply(lambda x: le.fit_transfor
```

In [ ]:

```
# test_final.info()
```

In [ ]:

```
# # number of clicks by IP
# ip_count = test_final.groupby('ip')['channel'].count().reset_index()
# ip_count.columns = ['ip', 'count_by_ip']
# ip_count.head()
```

In [ ]:

```
# merge this with the training data
# test_final = pd.merge(test_final, ip_count, on='ip', how='left')
```

In [ ]:

```
# del ip_count
```

In [ ]:

```
# test_final.info()
```

In [ ]:

```
# # predict on test data
# y_pred_test = model.predict_proba(test_final.drop('click_id', axis=1))
# y_pred_test[:10]
```

In [ ]:

```
# # # create submission file
# sub = pd.DataFrame()
# sub['click_id'] = test_final['click_id']
# sub['is_attributed'] = y_pred_test[:, 1]
# sub.head()
```

In [ ]:

```
# sub.to_csv('kshitij_sub_03.csv', float_format='%.8f', index=False)
```

In [ ]:

```
# # model

# dtrain = xgb.DMatrix(X_train, y_train)
# del X_train, y_train
# gc.collect()

# watchlist = [(dtrain, 'train')]
# model = xgb.train(params, dtrain, 30, watchlist, maximize=True, verbose_eval=1)
```

In [ ]:

```
# del dtrain
# gc.collect()
```

In [ ]:

```
# # Plot the feature importance from xgboost
# plot_importance(model)
# plt.gcf().savefig('feature_importance_xgb.png')
```

In [ ]:

```
# # Load the test for predict
# test = pd.read_csv(path+"test.csv")
```

In [ ]:

```
# test.head()
```

In [ ]:

```
# # number of clicks by IP
# ip_count = train_sample.groupby('ip')['channel'].count().reset_index()
# ip_count.columns = ['ip', 'count_by_ip']
# ip_count.head()
```

In [ ]:

```
# test = pd.merge(test, ip_count, on='ip', how='left', sort=False)
# gc.collect()
```

In [ ]:

```
# test = timeFeatures(test)
# test.drop(['click_time', 'datetime'], axis=1, inplace=True)
# test.head()
```

In [ ]:

```
# print(test.columns)
# print(train_sample.columns)
```

In [ ]:

```
# test = test[['click_id','ip', 'app', 'device', 'os', 'channel', 'day_of_week',
#        'day_of_year', 'month', 'hour', 'count_by_ip']]
```

In [ ]:

```
# dtest = xgb.DMatrix(test.drop('click_id', axis=1))
```

In [ ]:

```
# # Save the predictions
# sub = pd.DataFrame()
# sub['click_id'] = test['click_id']

# sub['is_attributed'] = model.predict(dtest, ntree_limit=model.best_ntree_limit)
# sub.to_csv('xgb_sub.csv', float_format='%.8f', index=False)
```

In [ ]:

```
# sub.shape
```

Summary

You have come a long way! Let's look at a summary of all that you have covered in the last two sessions.

In the first session, we went through the concepts of boosting and studied one of the earliest boosting algorithms, AdaBoost. We went through the process of updating the distribution of the data points by changing the probabilities attached to them and deciding the weights attached to the individual models that fit on this distribution.

In the next session, we went through Gradient Boosting and a modification of it, XGBoost. We were introduced to the intuition of gradient descent in the regression setting when we used the square loss function to close in on the residues. We developed from here on the Gradient Boosting algorithm in which each subsequent model trains on the gradients of the loss function. XGBoost follows the same procedure on trees with additional features like regularization and parallel tree learning algorithm for finding the best split.

You can download the lecture notes for this module from the link below:

Lecture Notes - Boosting Download

Question 1 Submitted

Report an error

PREVIOUS
Kaggle Practice Exercise

NEXT
Graded Questions

Supervised Learning - Boosting

In this module, you grasped the concepts of another supervised learning algorithm called Boosting. We learned some of the popular algorithms, namely **AdaBoost, Gradient Boosting** and a modification of Gradient Boosting, **XGBoost**.

Boosting

Boosting was first introduced in 1997 by Freund and Schapire in the popular algorithm, AdaBoost. It was originally designed for classification problems. Since its inception, many new boosting algorithms have been developed those tackle regression problems also and have become famous as they are used in the top solutions of many Kaggle competitions.

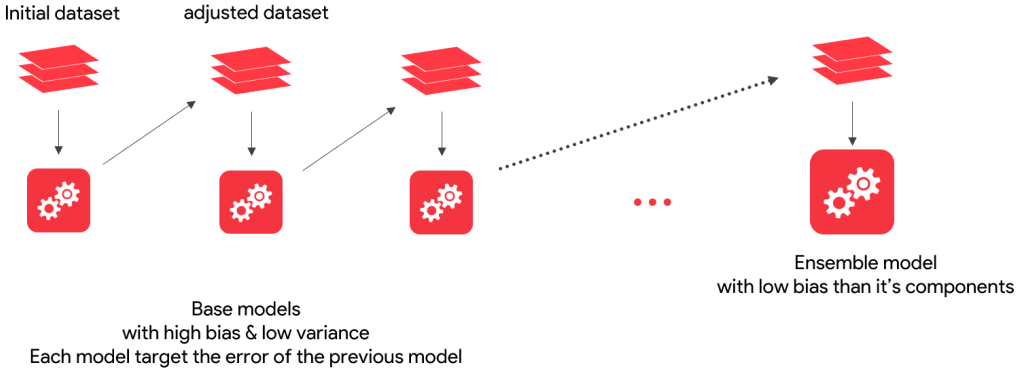
Let's start off with the basics of Boosting and move on to the boosting algorithms.

An ensemble is a collection of models which ideally should predict better than individual models. The key idea of boosting is to create an ensemble which makes high errors only on the less frequent data points.

Boosting leverages the fact that we can build a series of models specifically targeted at the data points which have been incorrectly predicted by the other models in the ensemble. If a series of models keep reducing the average error, we will have an ensemble having extremely high accuracy.

Boosting is a way of generating a strong model from a weak learning algorithm.

> A **weak learning algorithm** produces a model that does marginally better than a random guess. A random guess has a 50% chance of being right. Hence, any such model created by the weak learning algorithm shall have, say 60-70% chance of being correct.

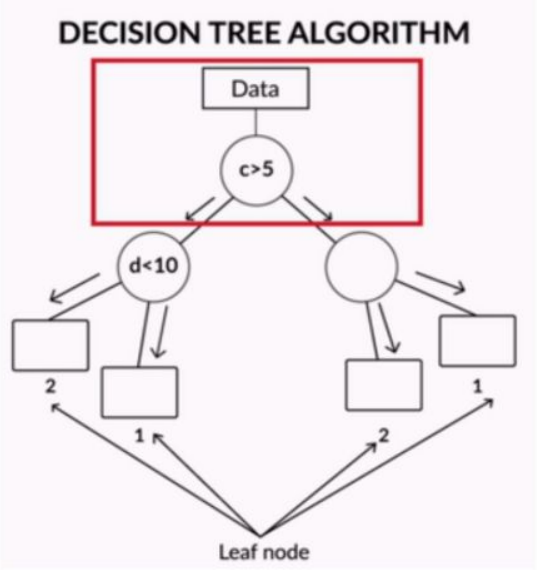


Boosting

There are other ways we can make the decision tree a weak learner like

1. max_depth: The maximum depth of the tree
2. min_samples_split: The minimum number of samples required to split an internal node
3. min_samples_leaf: The minimum number of samples required to be at a leaf node:
4. min_weight_fraction_leaf: The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node
5. max_leaf_nodes: The maximum number of leaf nodes that can be generated
6. min_impurity_decrease: A node will be split if this split induces a decrease of the impurity greater than or equal to this value
7. min_impurity_split: A node will split if its impurity is above the threshold, otherwise it is a leaf

The following picture delivers an example of a weak learner. We have a learning algorithm, decision tree. By restricting the depth of the tree, we can make this decision tree a weak learner. The red box indicates only one split which is also known as a stump.



The final objective is to create a strong model by making an ensemble of such weak models.

Here, it is important to understand the loss functions for regression and classification problems are different. Until now, we have defined the error function for a regression setting as the sum of squared difference between the actual and the predicted values while the misclassification rate for a classification problem.

Adaboost

AdaBoost stands for Adaptive Boosting, was developed by Schapire and Freund, who later on won the 2003 Godel Prize for their work. In this method, every subsequent model is built on a new distribution. This new distribution is created by changing the probability or weights attached with every point. Here, we explain the AdaBoost algorithm using the classification setting in which the target values are +1/-1.

At an iteration t, we have a distribution Dt of the training data T, with the data points having probability pd(t)(xi) on which we fit a model Ht and then use the results to create a new distribution Dt+1. The final model H, we build is an ensemble of all the individual models Hi with weights α i.

The AdaBoost process starts off with uniform distribution for all the points but as we move on to make additional models that add to the previous model, the distribution changes and hence, the objective function is expressed in terms of the probabilities of the different data points. The probabilities of the data points on which the next additional model is built are changed in such a way that the algorithm increases the probabilities of the points that were incorrectly classified by the current model and lowers the probabilities of the points that were correctly classified. With each new model, the distribution of the data changes.

There are essentially two steps involved in the AdaBoost algorithm:

1. Modify the current distribution to create a new distribution to generate a new model
2. Calculation of the weights given to each of the models to get the final ensemble

Here is the pseudo-code for Adaboost

1. Initialize the probabilities of the distribution as $\frac{1}{n}$ where n is the number of data points
2. For t = 0 to T, repeat the following (T is the total number of trees):
    1. Fit a tree h_t on the training data using the respective probabilities
    2. Compute $\epsilon_t = \sum_i^n D_i[h_t(x_i) \neq y_i]$
    3. Compute $\alpha_t = \frac{1}{2} ln(\frac{1-\epsilon_t}{\epsilon_t})$
    4. Update $D_{t+1}(i) = \frac{D_t(i) * e^{-\alpha_t y_i h_t(x_i)}}{z_t}$ where, $z_t = \sum_{i=1}^n D_i * e^{-\alpha_t y_i h_t(x_i)}$
3. Final Model: $H(x) = sign(\sum_{t=i}^T \alpha_t h_t(x))$

We can see that if the prediction is correct, yi.ht(xi) > 0 and since αt > 0(will look about this in the next section), pd(t+1)(xi) < pd(t)(xi).
Also, when the prediction is wrong, yi.ht(xi) < 0 and since αt > 0, pd(t+1)(xi) < pd(t)(xi).

In other words, every subsequent model we build after changing the distribution has a misclassification rate, here the error, $\epsilon_t < 0.5$. With this in mind, we can see that the $\alpha_t > 0$ as the error $\epsilon_t < 0.5$, $((1-\epsilon_t)/\epsilon_t)$ = positive and the ln(positive) > 0.

We continue this iteration until

- Low training error is achieved
- A preset number of weak learners have been added

We then make the final prediction by adding up the weighted prediction of every classifier. $H(x)=\text{sign}(\sum_{t=i}^{T} \alpha_t h_t(x))$

We can realize here that as we increase the number of trees/ iterations, the error will keep on decreasing. Before you apply the AdaBoost algorithm, you should specifically remove the Outliers. Since AdaBoost tends to boost up the probabilities of misclassified points and there is a high chance that outliers will be misclassified, it will keep increasing the probability associated with the outliers and make the progress difficult. Some of the ways to remove outliers are:

- Boxplots
- Cook's distance
- Z-score

Gradient Boosting

Gradient Boosting like AdaBoost trains many models in a gradual, additive, and sequential manner. But the major difference between the two is how they identify & handle the shortcomings of weak learners through loss functions.

To summarize here are the broader points on how does a GBM learn:

- We build the first weak learner using a sample from the training data; we will consider a decision tree as the weak learner or the base model. It may not necessarily be a stump, can grow a bigger tree but will still be weak i.e. still not be fully grown.
- Then the predictions are made on the training data using the decision tree just built.
- The gradient, in our case the residuals are computed and these residuals are the new response or target values for the next weak learner.
- A new weak learner is built with the residuals as the target values and a sample of observations from the original training data.
- Add the predictions obtained from the current weak learner to the predictions obtained from all the previous weak learners. The predictions obtained at each step are multiplied by the learning rate so that no

single model makes a huge contribution to the ensemble thereby avoiding overfitting. Essentially, with the addition of each weak learner, the model takes a very small step in the right direction.

- The next weak learner fits on the residuals obtained till now and these steps are repeated, either for a prespecified number of weak learners or if the model starts overfitting i.e. it starts to capture the niche patterns of the training data.
- GBM makes the final prediction by simply adding up the predictions from all the weak learners (multiplied by the learning rate)

Here is the pseudo-code for Gradient Boosting

At any iteration t, we repeat the following steps in the Gradient Boosting scheme of things:

1. Initialize a crude initial function F0 as $argmin \sum_{t=i}^{T} \boldsymbol{L}(\boldsymbol{y_i}, \hat{\boldsymbol{y}})$
2. For m = 1 to M (where M is the number of trees)
    1. Calculate the pseudo-residuals $r_{im} = -\frac{(\partial \boldsymbol{L}(\boldsymbol{y_i}, \boldsymbol{F}(\boldsymbol{x_i}))}{\partial \boldsymbol{F}(\boldsymbol{x_i})}$, where $F(x_i) = F_{m-1}(x_i)$, the pseudo residuals are the negative gradients for all data points
    2. Fit a base learner $h_m(x)$ to the pseudo-residuals, i.e train it using the training set $\sum_{i=1}^{n} (x_i, r_{im})$. Here the pseudo residuals are used as the response variable.
    3. Compute the step magnitude multiplier γ_m (in case of tree models, compute a γ_m different for every leaf/prediction)

    $\gamma_m = argmin \sum_{i=1}^{n} L(y_i, (F_{m-1} + \gamma * h_m(x_i))$
    4. Compute the next model $F_m = F_{m-1}(x_i) + \gamma_m * h_m(x_i)$
3. The final model is $F_M(x)$

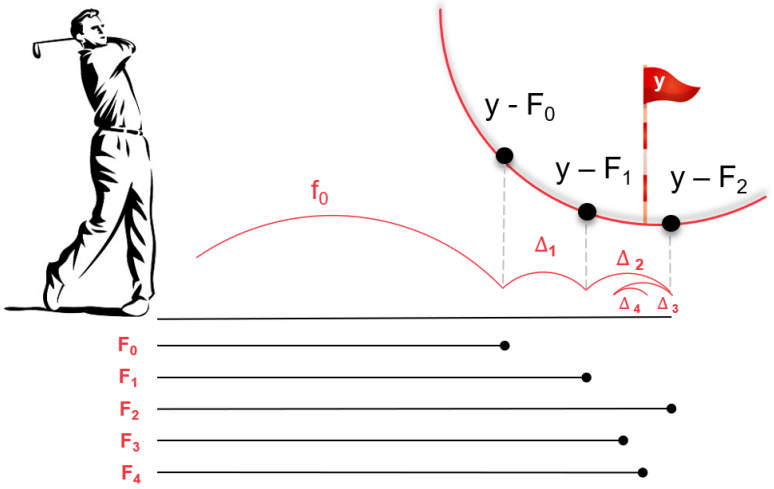
We see that the loss we get after fitting the model Fm is L(yi,Fm). In order to reduce this loss, we generate models Fm by adding an incremental model hm(xi) to Fm−1.

In other words, we select the hm such that L(yi,Fm−1) - L(y,Fm−1+hm) is the maximum.

The minimization is essentially a gradient descent problem. Hence, to find an h_m which when added to F_{m-1} reduces the loss, we take a step in the direction where the Loss $L(y_i,F_{m-1})$ reduces (with respect to F_{m-1}).

Mathematically, we take a step of size γ in the direction $-(\partial L(y_i,F(x_i))/\partial F(x_i)$, where $F(x_i)=F_{m-1}(x_i)$.

We stop when we see that the gradients are very close to zero.



Gradient Boosting

XGBoost

Extreme Gradient Boosting (XGBoost) is similar to the gradient boosting framework but more efficient and advanced implementation of the Gradient Boosting algorithm.

It was first developed by Taiqi Chen and became famous in solving the Higgs Boson problem. Due to its robust accuracy, it has been widely used in machine learning competitions as well. It uses more accurate approximations to tune the model and find the best fit.

Let's have a look at some of the advantages of XGBoost:

1. **Parallel Computing:** when you run xgboost, by default, it would use all the cores of your laptop/machine enabling its capacity to do parallel computation
2. **Regularization:** The biggest advantage of xgboost is that it uses regularization and controls the overfitting and simplicity of the model which gives it better performance.
3. **Enabled Cross-Validation:** XGBoost is enabled with internal Cross Validation function
4. **Missing Values:** XGBoost is designed to handle missing values internally. The missing values are treated in such a manner that if there exists any trend in missing values, it is captured by the model.
5. **Flexibility:** XGBoost is not just limited to regression, classification, and ranking problems, it supports user-defined objective functions as well. Furthermore, it supports user-defined evaluation metrics as well.

Because of parallel processing (speed) and model performance, we can say that XGBoost is gradient boosting on steroids.

Now, let's discuss some of the frequently used parameters that are used to regularize the Tree Boosting algorithms like the Gradient Tree Boosting and the XGBoost:

λ_t, the **learning rate,** is also known as **shrinkage.** It can be used to regularize the gradient tree boosting algorithm. λ_t typically varies from 0 to 1. Smaller values of λ_t lead to a larger value of a number of trees T (called *n_estimators* in the Python package XGBoost). This is because, with a slower learning rate, you need a larger number of trees to reach the minima. This, in turn, leads to longer training time.

On the other hand, if λ_t is large, we may reach the same point with a lesser number of trees (*n_estimators*), but there's the risk that we might actually miss the minima altogether (i.e. cross over it) because of the long stride we are taking at each iteration.

Some other ways of regularization are explicitly specifying the **number of trees T** and doing **subsampling**. Note that you shouldn't tune both λ_t and number of trees T together since a high λ_t implies a low value of T and vice-versa.

Subsampling is training the model in each iteration on a fraction of data (similar to how random forests build each tree). A typical value of subsampling is 0.5 while it ranges from 0 to 1. In random forests, subsampling is critical to ensure diversity among the trees, since otherwise, all the trees will start with the same training data and therefore look similar. This is not a big problem in boosting since each tree is any way built on the residual and gets a significantly different objective function than the previous one.

Disclaimer: All content and material on the upGrad website is copyrighted material, either belonging to upGrad or its bonafide contributors and is purely for the dissemination of education. You are permitted to access, print and download extracts from this site purely for your own education only and on the following basis:

- You can download this document from the website for self-use only.
- Any copy of this document, in part or full, saved to disk or to any other storage medium, may only be used for subsequent, self-viewing purposes or to print an individual extract or copy for non-commercial personal use only.
- Any further dissemination, distribution, reproduction, copying of the content of the document herein or the uploading thereof on other websites or use of the content for any other commercial/unauthorised purposes in any way which could infringe the intellectual property rights of upGrad or its contributors, is strictly prohibited.
- No graphics, images or photographs from any accompanying text in this document will be used separately for unauthorised purposes.
- No material in this document will be modified, adapted or altered in any way.
- No part of this document or upGrad content may be reproduced or stored in any other website or included in any public or private electronic retrieval system or service without upGrad's prior written permission.
- Any right not expressly granted in these terms are reserved.

Graded Questions

The following questions are **graded**.

Question 1/4

Mandatory

Gradient Boosting

Consider the following statements w.r.t Gradient Boosting and choose the correct:

1. At each iteration, we add an incremental model, which is fitted on the negative gradients of the loss function w.r.t the previous predictions.
2. We multiply λ_t(learning rate) with the incremental model h_{t+1} so that the new model doesn't overfit.

- Only 1
- Only 2
- Both 1 & 2 ✓ Correct

Feedback:

None of the above

Your answer is Correct.

Attempt 1 of 2

Continue

Report an error

Graded Questions

The following questions are **graded**.

Question 2/4

Mandatory

Adaboost

In Adaboost, each model has different say/importance according to the error it has made while predicting the training data.

This is depicted in the following equation: $\alpha = 0.5 \ln((1\text{-TOTAL ERROR})/(\text{TOTAL ERROR}))$

With respect to this, consider the following statements and choose the correct:

1. The classifier weight grows exponentially as the error approaches 0. Better classifiers are given exponentially more weight.
2. The classifier weight is 0.5 if the error rate is 0.5. This is because the classifier has only 50% accuracy.
3. The classifier weight grows exponentially negative as the error approaches 1. These types of classifiers are given a negative weight.

- 1 & 2 only
- 1 & 3 only ✓ Correct

2 & 3 only

All of the above

Your answer is Correct.

Attempt 2 of 2

Continue

Report an error

Graded Questions

The following questions are **graded**.

Question 3/4

Mandatory

General Expression for Residual

For a gradient boosting algorithm, let's say F_0 is the crude model with which we start off. For a model Ft which is fitted on the training data, the prediction we get for xi is Ft(xi). What is the general expression of the residuals generated once the Ft model is trained? Assume, y is the initial target variable.

- y - [Fo(xi)+F1(xi) + F2 (xi)+.....+ Ft−2(xi) + Ft−1(xi)]
- y - [F0(xi)+Ft−1(xi)]
- y - [Fo(xi)+F1(xi) + F2 (xi)+.....+ Ft−2(xi) + Ft−1(xi) + Ft(xi)] ✓ Correct

Feedback:

The residuals created by F1 is y−F0(xi)−F1(xi), for F2 it is y−F0(xi)−F1(xi) - F2(xi) and so on. In general, Ft trains on the residuals generated by the model Ft−1. So

Report an error

PREVIOUS
Summary

FINISH SESSION
Graded Questions

Graded Questions

The following questions are **graded**.

Question 4/4

Mandatory

XGBoost

Which of these features are the advantages of XGBoost algorithm?

More than one option can be correct.

- [x] Parallel and distributed computing ✓ Correct

  Feedback:
  Fast learning through parallel and distributed computing enables quicker model exploration.

- [ ] Optimization through first-order derivatives

- [x] Advanced regularisation ✓ Correct

  Feedback:

Report an error

PREVIOUS
Summary

FINISH SESSION
Graded Questions

Mathematics behind XGBoost

The upcoming text discusses the maths behind the XGBoost. It is expected that you write the equations to have a better understanding.

NOTE: For a detailed understanding, please attend the live session on Saturday.
Both XGBoost and GBM follow the principle of gradient boosted trees, but XGBoost uses a more **regularised model formulation to control overfitting**, which gives it better performance, which is why it's also known as '**regularised boosting**' technique.

The mathematics behind the XGBoost model:

Note: Here unlike the added model represented earlier by h_m, here we are representing the model by h_t at the t^{th} iteration.
In an ideal machine learning model, the objective function is a sum of Loss function "L" and regularization "Ω". Loss function controls the predictive power of the algorithm and regularization controls its simplicity

$$\textbf{Objective Function : Training Loss}(\mathbf{L}) + \textbf{Regularization}(\mathbf{\Omega})$$

The algorithm we have seen in Gradient Boosting has the objective function as only the Training Loss while the XGBoost objective function constitutes of loss function evaluated

$i=1$ $i=1$

Where h_t means predictions coming from the t^{th} tree.

From Gradient Boosting we have understood that the final model will be represented by:

$$F_t(x_i) = F_0(x_i) + \sum_{t=1}^{T} h_t(x_i) = F_{t-1}(x_i) + h_t(x_i)$$

Here $F_t(x_i)$ is the prediction of the i^{th} instance x_i at the t^{th} iteration and to calculate this we need to add h_t to our previous prediction. So now we will apply this to our objective function

$$\begin{aligned}\text{obj}^{(t)} &= \sum_{i=1}^{n} L(y_i, F_t(x_i)) + \sum_{t=1}^{T} \Omega(h_t) \\ &= \sum_{i=1}^{n} L(y_i, F_{t-1}(x_i) + h_t(x_i)) + \sum_{t=1}^{T} \Omega(h_t)\end{aligned}$$

Here we greedily add the h_t that improves our model the most by minimizing our objective function. It acts as a **small update** to our final model.

In Gradient boosting algorithm we obtained $F_t(x_i)$ at each iteration by **fitting a base learner to the negative gradient of the loss function with respect to previous iteration's value.**

In XGBoost, we explore several base learners/models and pick a model that minimizes the loss.

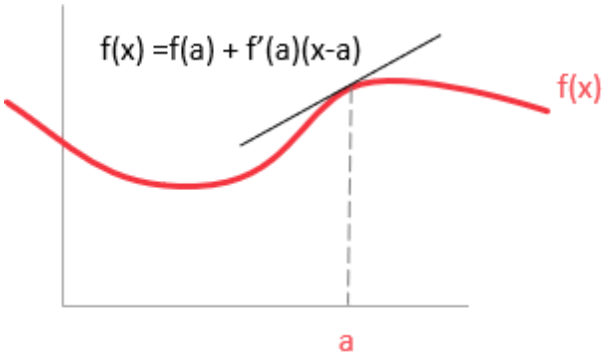
Our model h_t in XGBoost contains the structure of the tree and leaf scores, thus which seems a fairly complex optimization problem to be dealt with "Gradient descent' technique. This is due to the fact that an ensemble model includes "functions" as

value of the loss function for a base learner

Taylor series approximation of the loss

The above described objective function can be approximated using Taylor series expansion and hence can be solved.

To have a better understanding, let's take a foot back and recall Taylor series from our calculus class.



Taylor approximation of a function around point a.

A Taylor series is a series expansion of a function about a point, Suppose that the function f(x) is infinitely differentiable(smooth) at x = a.
For **f(x)**, **Δx** is the new learner we added in step **t** and **a** is the prediction at step (**t-1**)
Δx = (**x-a**) is the new learner that we need to add in step (t) in order to greedily minimize the objective.

By Taylor's expansion, any continuous function can be approximated by the linear combination of its first-order gradient and the quadratic function of the second-order

which can also be rephrased as:

$$f(x+h) = f(x) + \frac{f'(x)}{1!}(h) + \frac{f''(x)}{2!}(h)^2 + \frac{f'''(x)}{3!}(h)^3 + \cdots,$$

by replacing a with x and $x - a$ with h, where the function is continuous and n times differentiable in an interval $[x, x+h]$

Question 1/2 Mandatory

Taylor Series

The Taylor series for the functions $x^4 + x - 2$ which is differentiable at a =1 about the specified a is:

- $5(x+1) + 6(x-1)^2 + 4(x+1)^3 + (x-1)^4$
- $5(x-1) + 6(x-1)^2 + 4(x-1)^3 + (x-1)^4$ ✓ Correct

Feedback:

f (x) = $x^4 + x - 2$.

$f'(x) = 4x^3 + 1$

$f''(x) = 12x^2$

$f'''(x) = 24x$

which is $5(x-1)+6(x-1)^2+4(x-1)^3+(x-1)^4$

$(x-1)+(x-1)^2+(x-1)^3+(x-1)^4$

None of these

Your answer is Correct. Attempt 1 of 2 Continue

So, in this case, we take Taylor expansion of the loss function up to the second order only and approximating the later differentiation terms to be ~ 0.

This will result in the objective function

$$\text{obj}^{(t)} = \sum_{i=1}^{n} L(y_i, F_{t-1}(x_i) + h_t(x_i)) + \sum_{t=1}^{T} \Omega(h_t)$$

as:

$$\text{obj}^{(t)} = \sum_{i=1}^{n} [L(y_i, F_{t-1}(x_i)) + p_i h_t(x_i) + \frac{1}{2} q_i h_t^2(x_i)] + \Omega(h_t)$$

where the p_i and q_i are defined as

$$p_i = \partial_{F_{t-1}(x_i)} L(y_i, F_{t-1}(x_i))$$
$$q_i = \partial^2_{F_{t-1}(x_i)} L(y_i, F_{t-1}(x_i))$$

Hence, this becomes our optimization goal for the new tree. The advantage of this definition is that it only depends on p_i and q_i which also lets XGBoost support custom loss functions including logistic regression, weighted logistic regression etc.

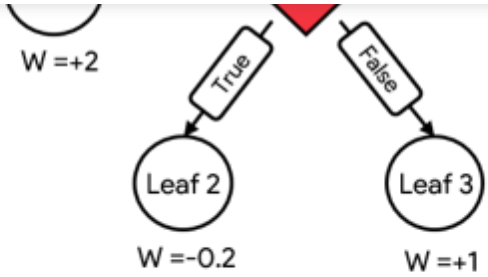
XGBoost Regularisation

We have introduced the training step here, now what remained is defining the regularization term. Let's define the complexity of the tree $\Omega(h_t)$. In order to do so, let us first go through the definition of the tree $h_t(x)$

$$h_t(x) = w_{m(x)}, w \in R^T, m : R^d \rightarrow \{1, 2, \cdots, J_T\}$$

Here w is the vector of scores on leaves, m is a function (structure) assigning each data point to the corresponding leaf, and J_T is the number of leaves. Now the regularization function becomes:

$$\Omega(h_t) = \gamma * T + \frac{1}{2} * \lambda \sum_{j=1}^{T} w_j^2$$



$$\Omega(h_t) = \Upsilon * 3 + \frac{1}{2}\lambda(4 + 0.04 + 1)$$

An example of the complexity function for a tree carrying 3 leaves

Further diving deep into XGBoost

Upon defining, $I_j = \{i|m(x_i) = j\}$ as the instance set of leaf j and imputing the training and regularization part in the objective function, we can write it with the t-th tree as:

$$\begin{aligned}\text{obj}^{(t)} &\approx \sum_{i=1}^{n}[p_i w_{m(x_i)} + \frac{1}{2}q_i w^2_{m(x_i)}] + \gamma T + \frac{1}{2}\tau\sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T}[(\sum_{i\in I_j} p_i)w_j + \frac{1}{2}(\sum_{i\in I_j} q_i + \tau)w_j^2] + \gamma T\end{aligned}$$

Note: In the second line, the index of the summation has been changed because all the data points on the same leaf get the same score.

$obj = \sum_{j=1}[P_j w_j + \frac{1}{2}(Q_j + \tau)w_j] + \gamma T$

As w_js are independent with respect to each other and $P_j w_j + \frac{1}{2}(Q_j + \tau)w_j^2$ is a quadratic function hence for a fixed structure $m(x)$, we can compute the optimal weight w_j^* of leaf j as:

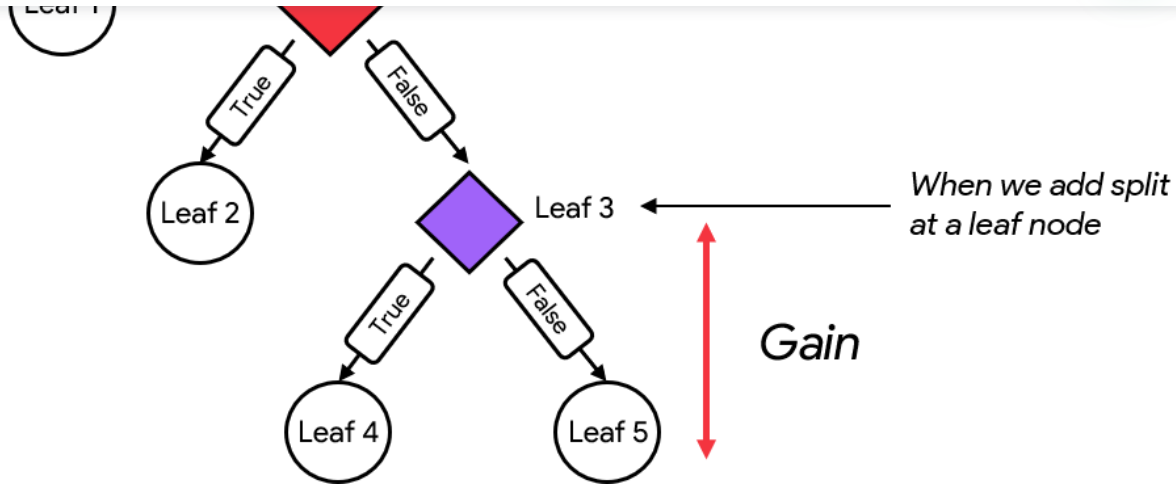
$$w_j^* = -\frac{P_j}{Q_j + \tau}$$

by differentiating the function $P_j w_j + \frac{1}{2}(Q_j + \tau)w_j^2$ with respect to w_j.
And, calculate the corresponding optimal value of the objective function by substituting the above-found w_j^* as

$$obj^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{P_j^2}{Q_j + \tau} + \gamma T$$

Note: The scoring function can be used as a measure of how good a tree structure $m(x)$ is.

It may sound a bit complicated but basically, for a given tree structure, we push the statistics
p_i and q_i to the leaves they belong to and use the formula to calculate how good the tree is.



$$\text{obj} = -\frac{1}{2}\left[\frac{P_1^2}{(Q_1+\lambda)} + \frac{P_2^2}{(Q_2+\lambda)} + \frac{P_3^2}{(Q_3+\lambda)}\right] + \Upsilon * 3$$

$$\text{obj} = -\frac{1}{2}\left[\frac{P_1^2}{(Q_1+\lambda)} + \frac{P_2^2}{(Q_2+\lambda)} + \frac{P_4^2}{(Q_4+\lambda)} + \frac{P_5^2}{(Q_5+\lambda)}\right] + \Upsilon * 4$$

$$Gain = \frac{1}{2}\left[\frac{P_4^2}{(Q_4+\lambda)} + \frac{P_5^2}{(Q_5+\lambda)} - \frac{P_3^2}{(Q_3+\lambda)}\right] - \Upsilon$$

$$Gain = \frac{1}{2}\left[\frac{P_L^2}{(Q_L+\lambda)} + \frac{P_R^2}{(Q_R+\lambda)} - \frac{P_L^2+P_R^2}{(Q_L+QR+\lambda)}\right] - \Upsilon$$

Based on some split criteria, the node is divided into left and right branches. Some instances fall in the left node and other instances fall in the right leaf node.

XGBoost greedily builds a tree. The split that results in maximum loss reduction or gain is chosen.

Gain= $Loss$(parent) - [$Loss_L$(left branch)+$Loss_R$(right branch)]

reading material below.

Additional References:

- Tianqi Chen explaining the XGBoost
- XGBoost: A Scalable Tree Boosting System
- Finding the weights of a decision tree
- Complete Guide to Parameter Tuning in XGBoost
- You can go through this paper for learning more about the Parallel Tree Learning Algorithm used for finding best split authored by the founder Tianqi Chen
- To have a better understanding of the objective function in the XGBoost scheme of things you can read from pg. 31 in the following pdf
- To read more about CatBoost, go through the following paper and the following site
- To read more about LightGBM which uses GOSS, refer the following paper and original documentation

Report an error

FINISH SESSION
Mathematics behind XGBoost

which can also be rephrased as:

$$f(x+h) = f(x) + \frac{f'(x)}{1!}(h) + \frac{f''(x)}{2!}(h)^2 + \frac{f'''(x)}{3!}(h)^3 + \cdots,$$

by replacing a with x and $x - a$ with h, where the function is continuous and n times differentiable in an interval $[x, x+h]$

Question 2/2 Mandatory

Taylor Series

Given $f(3) = 6, f'(3) = 8, f''(3) = 11$, and all other higher order derivatives of $f(x)$ are zero at x = 3, and assuming the function and all its derivatives exist and are continuous between x = 3 and x = 7, the value of $f(7)$ is

- 42
- 87.5
- 126 ✓ Correct

Feedback:
The Taylor series is given by,

Upon substituting the respective values we'll get 126

231.5 ✕ Incorrect

Feedback:
The Taylor series is given by,

$$f(x+h) = f(x) + \frac{f'(x)}{1!}(h) + \frac{f''(x)}{2!}(h)^2 + \frac{f'''(x)}{3!}(h)^3 + \cdots,$$

$x = 3, h = 7 - 3 = 4$

$$f(7) = f(3) + f'(3).4 + \frac{f''(3)}{2}.4^2 + \frac{f'''(3)}{4}.4^3$$

Upon substituting the respective values what we'll get?

Your answer is Wrong. Attempt 2 of 2 Continue

So, in this case, we take Taylor expansion of the loss function up to the second order only and approximating the later differentiation terms to be ~ o.

This will result in the objective function

$$\text{obj}^{(t)} = \sum_{i=1}^{n} L(y_i, F_{t-1}(x_i) + h_t(x_i)) + \sum_{t=1}^{T} \Omega(h_t)$$

as:

$$\text{obj}^{(t)} = \sum_{i=1}^{n} [L(y_i, F_{t-1}(x_i)) + p_i h_t(x_i) + \frac{1}{2} q_i h_t^2(x_i)] + \Omega(h_t)$$