# TITLE: PREDICTING HOUSE PRICES USING MACHINE LEARNING

**Phase 2: Innovation**

**Consider exploring advanced regression techniques like Gradient Boosting or XGBoost for improved prediction accuracy**

**Abstract:** House price fluctuates each and every year due to changes in land value and change in infrastructure in and around the area. Centralised system should be available for prediction of house price in correlation with neighbourhood and infrastructure, will help customer to estimate the price of the house. Also, it assists the customer to come to a conclusion where to buy a house and when to purchase the house. Different factors are taken into consideration while predicting the worth of the house like location, neighbourhood and various amenities like garage space etc. Developing a model starts with Pre-processing data to remove all sort of discrepancies and fill null values or remove data outliers and make data ready to be processed. The categorical attribute can be converted into required attributes using one hot encoding methodology. Later the house price is predicted using XGBoost regression technique. Keywords: Machine Learning; XGBoost Regression; Gradient Boost; Ensemble Learning; House Price Prediction.

**Introduction:**

Using advanced regression techniques like Gradient Boosting and XGBoost can significantly enhance prediction accuracy when it comes to predicting house prices using machine learning. These techniques are particularly beneficial because they are capable of handling complex relationships within the data and mitigating issues like overfitting.

**Gradient Boosting:**

Gradient Boosting is an ensemble learning method that combines the predictions of multiple weak learners (typically decision trees) to create a strong predictive model. It's known for its robustness and can adapt to various data types and characteristics.

**XGBoost :**

XGBoost, or Extreme Gradient Boosting, is an optimized and efficient implementation of the Gradient Boosting algorithm. It is widely favored in data science competitions due to its speed and high predictive accuracy. XGBoost offers various hyperparameters to fine-tune the model and improve performance.

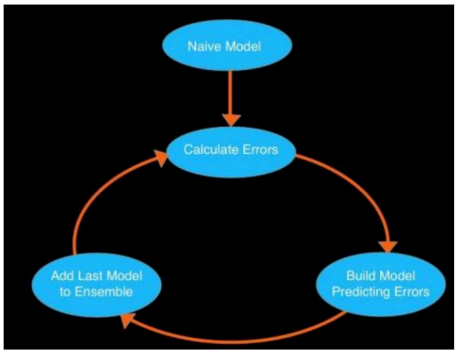
**What is xGBoost?**

XGBoost is the leading model for working with standard tabular data (the type of data you store in Pandas Data Frames, as opposed to more exotic types of data like images and videos). XGBoost models dominate many Kaggle competitions.

To reach peak accuracy, XGBoost models require more knowledge and model tuning than techniques like Random Forest. After this tutorial, you'ill be able to

- Follow the full modeling workflow with XGBoost

- Fine-tune XGBoost models for optimal performance xGBoost is an implementation of the Gradient Boosted Decision Trees algorithm (scikit-learn has another version of this algorithm, but XGBoost has some technical advantages.) What is Gradient Boosted Decision Trees? We'll walk through a diagram.




We go through cycles that repeatedly builds new models and combines them into an ensemble model. We start the cycle by calculating the errors for each observation in the dataset. We then build a new model to predict those. We add predictions from this error-predicting model to the "ensemble of models."

To make a prediction, we add the predictions from all previous models. We can use these predictions to calculate new errors, build the next model, and add it to the ensemble.

There's one piece outside that cycle. We need some base prediction to start the cycle. In practice, the initial predictions can be pretty naive. Even if it's predictions are wildly inaccurate, subsequent additions to the ensemble will address those errors.

XG Boost Regression Algorithm for House Price Prediction

Input: House attributes dataset.

Dataset Link: https://www.kaggle.com/datasets/vedavyasv/usa.housing

Output: Price of house.

1. Check input dataset for missing values and calculate d mean is replaced in place of missing value.

2. Divide attributes based on values in data fields as categorical and non-categorical rows.

3. Check Non categorical rows for outliers using outlier detection techniques and remove all outliers

4. Convert categorical rows into binary vectors using one hot encoding.

5. Divide dataset for cross validation using train test split.

6. Apply Ensemble learning through training and combining individual models termed as buse learners in order to derive a single prediction.

a) Calculate Mean Squared Error (MSE) with true values to predicted values.

b) Classify independent models as weak-learners and strong-learners using error detection

c) Total mean cancels bad prediction with good prediction.

7. Objective function contains the loss function and difference between actual value and predicted value

This process may sound complicated, but the code to use it is straightforward. We'll fill in some additional explanatory details in the model tuning section below.

**Example**

We will start with the data pre-loaded into train_X, test_X, train_y, test_y.

In [1]:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Imputer
data = pd.read_csv('../input/train.csv')
data.dropna(axis=0, subset=['SalePrice'], inplace=True)
y = data.SalePrice
X = data.drop(['SalePrice'], axis=1).select_dtypes(exclude=['object'])
train_X, test_X, train_y, test_y = train_test_split(X.as_matrix(), y.as_matrix(), test_size=0.25)
my_imputer = Imputer()
train_X = my_imputer.fit_transform(train_X)
test_X = my_imputer.transform(test_X)
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:9: FutureWarning: Method .as_matrix will be removed in a future version. Use .values instead.
  if __name__ == '__main__':
/opt/conda/lib/python3.6/site-packages/sklearn/utils/deprecation.py:58: DeprecationWarning: Class Imputer is deprecated; Imputer was deprecated in version 0.20 and will be removed in 0.22. Import impute.SimpleImputer from sklearn instead.
  warnings.warn(msg, category=DeprecationWarning)
```

We build and fit a model just as we would in scikit-learn.

In [2]:

```
from xgboost import XGBRegressor
my_model = XGBRegressor()
```

```
# Add silent=True to avoid printing out updates with each cycle
my_model.fit(train_X, train_y, verbose=False)
unfold_moreShow hidden output
```

We similarly evaluate a model and make predictions as we would do in scikit-learn.

```
In [3]:
# make predictions
predictions = my_model.predict(test_X)
from sklearn.metrics import mean_absolute_error
print("Mean Absolute Error : " + str(mean_absolute_error(predictions, test_y)))
Mean Absolute Error : 17543.750299657535
```

**Model Tuning**

XGBoost has a few parameters that can dramatically affect your model's accuracy and training speed. The first parameters you should understand are:

n_estimators and early_stopping_rounds

n_estimators specifies how many times to go through the modeling cycle described above.

In the underfitting vs overfitting graph, n_estimators moves you further to the right. Too low a value causes underfitting, which is inaccurate predictions on both training data and new data. Too large a value causes overfitting, which is accurate predictions on training data, but inaccurate predictions on new data (which is what we care about). You can experiment with your dataset to find the ideal. Typical values range from 100-1000, though this depends a lot on the learning rate discussed below.

The argument early_stopping_rounds offers a way to automatically find the ideal value. Early stopping causes the model to stop iterating when the validation score stops improving, even if we aren't at the hard stop for n_estimators. It's smart to set a high value for n_estimators and then use early_stopping_rounds to find the optimal time to stop iterating.

Since random chance sometimes causes a single round where validation scores don't improve, you need to specify a number for how many rounds of straight deterioration to allow before stopping. early_stopping_rounds = 5 is a reasonable value. Thus we stop after 5 straight rounds of deteriorating validation scores.

Here is the code to fit with early_stopping:

```
my_model = XGBRegressor(n_estimators=1000)
my_model.fit(train_X, train_y, early_stopping_rounds=5,
             eval_set=[(test_X, test_y)], verbose=False)
XGBRegressor(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=1000,
    n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
    reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
    silent=True, subsample=1)
```

When using early_stopping_rounds, you need to set aside some of your data for checking the number of rounds to use. If you later want to fit a model with all of your data, set n_estimators to whatever value you found to be optimal when run with early stopping.

**learning_rate**

Here's a subtle but important trick for better XGBoost models:

Instead of getting predictions by simply adding up the predictions from each component model, we will multiply the predictions from each model by a small number before adding them in. This means each tree we add to the ensemble helps us less. In practice, this reduces the model's propensity to overfit.

So, you can use a higher value of n_estimators without overfitting. If you use early stopping, the appropriate number of trees will be set automatically.

In general, a small learning rate (and large number of estimators) will yield more accurate XGBoost models, though it will also take the model longer to train since it does more iterations through the cycle.

Modifying the example above to include a learing rate would yield the following code:

```
my_model = XGBRegressor(n_estimators=1000, learning_rate=0.05)
my_model.fit(train_X, train_y, early_stopping_rounds=5,
             eval_set=[(test_X, test_y)], verbose=False)
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
    colsample_bytree=1, gamma=0, learning_rate=0.05, max_delta_step=0,
    max_depth=3, min_child_weight=1, missing=None, n_estimators=1000,
    n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
    reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
    silent=True, subsample=1)
```

**n_jobs**

On larger datasets where runtime is a consideration, you can use parallelism to build your models faster. It's common to set the parameter n_jobs equal to the number of cores on your machine. On smaller datasets, this won't help.

The resulting model won't be any better, so micro-optimizing for fitting time is typically nothing but a distraction. But, it's useful in large datasets where you would otherwise spend a long time waiting during the fit command.

XGBoost has a multitude of other parameters, but these will go a very long way in helping you fine-tune your XGBoost model for optimal performance.

**Conclusion**

XGBoost is currently the dominant algorithm for building accurate models on conventional data (also called tabular or strutured data). Go apply it to improve your models.