1. Introduction

If we have some background in machine learning and we'd like to learn how to quickly improve the quality of our models. In this course, we will accelerate our machine learning expertise by learning how to :

- tackle data types often found in real-world datasets(missing values, categorical variables).
- design pipleine to improve the quality of our machine learning code.
- use advanced techniques for model validation(CV)
- build state-of-the-art that are widely used to win Kaggle competiotns(XGBoost).
- Avoid common and important data science mistakes(leakage).

In [1]:
```python
# Progress of Machine Learning

# Preprocessing

import pandas as pd
from sklearn.model_selection import train_test_split

# Read the data
X_full = pd.read_csv('../../KAGGLE/Kaggle_House_Price/train.csv', index_col='Id')
X_test_full = pd.read_csv('../../KAGGLE/Kaggle_House_Price/test.csv', index_col='Id')

# Obtain target and predictors
y = X_full.SalePrice
features = ['LotArea', 'YearBuilt', '1stFlrSF', '2ndFlrSF', 'FullBath', 'BedroomAbvGr', 'TotRmsAbvGrd']
X = X_full[features].copy()
X_test = X_test_full[features].copy()

# Break off validation set from training data
X_train, X_valid, y_train, y_valid = train_test_split(X, y, train_size=0.8, test_size=0.2,
                                                      random_state=0)

# Gridsearch
from sklearn.ensemble import RandomForestRegressor

# Define the models
model_1 = RandomForestRegressor(n_estimators=50, random_state=0)
model_2 = RandomForestRegressor(n_estimators=100, random_state=0)
model_3 = RandomForestRegressor(n_estimators=100, criterion='mae', random_state=0)
model_4 = RandomForestRegressor(n_estimators=200, min_samples_split=20, random_state=0)
model_5 = RandomForestRegressor(n_estimators=100, max_depth=7, random_state=0)

models = [model_1, model_2, model_3, model_4, model_5]

# Make best model

best_model = RandomForestRegressor(n_estimators=100, criterion='mae', random_state=0)

# Fit the model to the training data
best_model.fit(X, y)

# Generate test predictions
preds_test = best_model.predict(X_test)

# Save predictions in format used for competition scoring
output = pd.DataFrame({'Id': X_test.index,
                       'SalePrice': preds_test})
output.to_csv('submission.csv', index=False)
```

2. Submit your results

Once you have successfully completed Step 2, you're ready to submit your results to the leaderboard! First, you'll need to join the competition if you haven't already. So open a new window by clicking on this link. Then click on the **Join Competition** button. *(If you see a "Submit Predictions" button instead of a "Join Competition" button, you have already joined the competition, and don't need to do so again.)*

Next, follow the instructions below:

1. Begin by clicking on the **Save Version** button in the top right corner of the window. This will generate a pop-up window.
2. Ensure that the **Save and Run All** option is selected, and then click on the **Save** button.
3. This generates a window in the bottom left corner of the notebook. After it has finished running, click on the number to the right of the **Save Version** button. This pulls up a list of versions on the right of the screen. Click on the ellipsis **(...)** to the right of the most recent version, and select **Open in Viewer**. This brings you into view mode of the same page. You will need to scroll down to get back to these instructions.
4. Click on the **Output** tab on the right of the screen. Then, click on the file you would like to submit, and click on the **Submit** button to submit your results to the leaderboard.

You have now successfully submitted to the competition!

If you want to keep working to improve your performance, select the **Edit** button in the top right of the screen. Then you can change your code and repeat the process. There's a lot of room to improve, and you will climb up the leaderboard as you work.

3. Missing values

1. Three Approches

   1) A simple options : Drop Columns with Missing Values

   The simplest option is to drop columns with missing values. Unless most values in the dropped columns are missing, the model loses access to a lot of information with this approach.
   As an extreme example, consider a dataset with 10,000 rows, where one important column is missing a single entry. This approach would rop the column entirely.

   2) A Better option : Imputation

   Imputation fills in the missing values with some number. For instance, we can fill in the mean value along each column. The imputed value won't be exactly right in most cases, but it usually leads to more accurate models than you would get from dropping the column entirely.

   3) An Extension to Imputation

   Imputation is the standard approach, and it usually works well. However, imputed values may be systematically above or below thier actual values. Or rows with missing values may be unique in some other way. In that case, your model would make better predictions by considering which values were originally missing. In this approach, we imputed the missing values, as before. And, additionally, for each column with missing entries in the original dataset, we add a new column that shows the location of the imputed entries.

In [2]:
```python
import pandas as pd
from sklearn.model_selection import train_test_split

# Load the data
data = pd.read_csv('../../KAGGLE/Kaggle_House_Price/train.csv')

# Select target
y = data.SalePrice

# To keep things simple, we'll use only numerical predictors
melb_predictors = data.drop(['SalePrice'], axis=1)
X = melb_predictors.select_dtypes(exclude=['object'])

# Divide data into training and validation subsets
X_train, X_valid, y_train, y_valid = train_test_split(X, y, train_size=0.8, test_size=0.2,
                                                      random_state=0)


from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error

# Function for comparing different approaches
def score_dataset(X_train, X_valid, y_train, y_valid):
    model = RandomForestRegressor(n_estimators=10, random_state=0)
    model.fit(X_train, y_train)
    preds = model.predict(X_valid)
    return mean_absolute_error(y_valid, preds)
```

In [3]:
```python
# Get names of columns with missing values
cols_with_missing = [col for col in X_train.columns if X_train[col].isnull().any()]

# Drop columns in training and validation data
reduced_X_train = X_train.drop(cols_with_missing, axis=1)
reduced_X_valid = X_valid.drop(cols_with_missing, axis=1)

print("MAE from Approach 1 (Drop columns with missing values):")
print(score_dataset(reduced_X_train, reduced_X_valid, y_train, y_valid))


from sklearn.impute import SimpleImputer

# Imputation
my_imputer = SimpleImputer()
imputed_X_train = pd.DataFrame(my_imputer.fit_transform(X_train))
imputed_X_valid = pd.DataFrame(my_imputer.transform(X_valid))

# Imputation removed column names; put them back
imputed_X_train.columns = X_train.columns
imputed_X_valid.columns = X_valid.columns

print("MAE from Approach 2 (Imputation):")
print(score_dataset(imputed_X_train, imputed_X_valid, y_train, y_valid))

# Make copy to avoid changing original data (when imputing)
X_train_plus = X_train.copy()
X_valid_plus = X_valid.copy()

# Make new columns indicating what will be imputed
for col in cols_with_missing:
    X_train_plus[col + '_was_missing'] = X_train_plus[col].isnull()
    X_valid_plus[col + '_was_missing'] = X_valid_plus[col].isnull()

# Imputation
my_imputer = SimpleImputer()
imputed_X_train_plus = pd.DataFrame(my_imputer.fit_transform(X_train_plus))
imputed_X_valid_plus = pd.DataFrame(my_imputer.transform(X_valid_plus))

# Imputation removed column names; put them back
imputed_X_train_plus.columns = X_train_plus.columns
imputed_X_valid_plus.columns = X_valid_plus.columns

print("MAE from Approach 3 (An Extension to Imputation):")
print(score_dataset(imputed_X_train_plus, imputed_X_valid_plus, y_train, y_valid))
```

```
MAE from Approach 1 (Drop columns with missing values):
19003.3198630137
MAE from Approach 2 (Imputation):
19243.57294520548
MAE from Approach 3 (An Extension to Imputation):
19349.617465753425
```

4. Categorical Variables

1. Introduction :
   A categorical variables takes only a limited number of values. Consider a surve that asks how often you eat breakfast and provides four options : "Never", "Rarely", "Most days", or "Every day". In this case, the data is categorical, because responses fall into a fixed set of categories. If people responded to a survey about which brand of car they owned, the responses would fall into categorical like "Honda", "Toyota", and "Ford". In this case, te data is also categorical.
2. Three Approcaches :

   1) Drop Categorical Variables :
   The easiest approach to dealing with categorical variables is to simply remove them from the dataset. This approach will only work well if the columns did not contain useful information.

   2) Ordinal Encoding :
   Ordinal encoding assigns each unique value to a different integer. This approach assumes an ordering of the categories : "Never"(0) < "Rarely"(1) < "Most days"(2) < "Every day"(3). This assumption makes sense in this example, because there is an indisputable ranking to the categories. Not all categorical variables have a clear ordering in the values, but we refer to those that do as ordinal variables. For tree-based models, you can expect encoding to work well with ordinal variables.

   3) One-Hot encoding :
   One-hot encoding creates new columns indicating the presence of each possible value in the original data. To understand this, we'll work through an example. In the original dataset, "Color" is a categorical variable with three categories : "Red", "Yellow", and "Green". The corresponding one-hot encoding contains one column for each possible value, and one row for each row in the original dataset. Wherever the original value was "Red", we put a 1 in the "Red" column; If the original value was "Yellow", we put a 1 in the "Yellow" column, and so on.

   In contrast to ordinal encoding, one-hot encoding does not assume an ordering of the categories. Thus you can expect this pproach to work particulary well if there is no clear ordering in teh categoricla data. We refer to categorical variables without an instinc ranking as nominal variables.

   One-hot encoding generally does not perform well if the categorical variable takes on a large number of values generally won't use it for variables taking more than 15 different values.

순서형 자료의 경우(n > 10) : replace, 명목형 자료의 경우(n < 10) : get dummies

In [4]:
```python
import pandas as pd
from sklearn.model_selection import train_test_split

# Load the data
data = pd.read_csv('../../KAGGLE/Kaggle_House_Price/train.csv')

# Separate target from predictors

y = data.SalePrice
X = data.drop(['SalePrice'], axis = 1)

# Divide data into training and validation subsets
X_train_full, X_valid_full, y_train, y_valid = train_test_split(X, y, train_size=0.8, test_size=0.2,
                                                                random_state=0)

# Drop columns with missing values (simplest approach)
cols_with_missing = [col for col in X_train_full.columns if X_train_full[col].isnull().any()]
X_train_full.drop(cols_with_missing, axis=1, inplace=True)
X_valid_full.drop(cols_with_missing, axis=1, inplace=True)

# "Cardinality" means the number of unique values in a column
# Select categorical columns with relatively low cardinality (convenient but arbitrary)
low_cardinality_cols = [cname for cname in X_train_full.columns if X_train_full[cname].nunique() < 10 and
                        X_train_full[cname].dtype == "object"]

# Select numerical columns
numerical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].dtype in ['int64', 'float64']]

# Keep selected columns only
my_cols = low_cardinality_cols + numerical_cols
X_train = X_train_full[my_cols].copy()
X_valid = X_valid_full[my_cols].copy()
```

In [5]:
```python
X_train.head()
```

Out[5]:

	MSZoning	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	Condition1	Condition2	BldgType	HouseStyle	RoofStyle	RoofMatl	ExterQua
618	RL	Pave	Reg	Lvl	AllPub	Inside	Gtl	Norm	Norm	1Fam	1Story	Hip	CompShg	E
870	RL	Pave	Reg	Lvl	AllPub	Inside	Gtl	PosN	Norm	1Fam	1Story	Hip	CompShg	T.
92	RL	Pave	IR1	HLS	AllPub	Inside	Gtl	Norm	Norm	1Fam	1Story	Gable	CompShg	T.
817	RL	Pave	IR1	Lvl	AllPub	CulDSac	Gtl	Norm	Norm	1Fam	1Story	Hip	CompShg	G
302	RL	Pave	IR1	Lvl	AllPub	Corner	Gtl	Norm	Norm	1Fam	1Story	Gable	CompShg	G

Next, we obtain a list of all of the categorical variables in the training data.

We do this by checking the data type of each column. The object dtype indicates a column has text. For this dataset, the columns with text indicate categorical variables.

In [6]:
```python
# Get list of categorical variables
s = (X_train.dtypes == 'object')
object_cols = list(s[s].index)

print("Categorical variables:")
print(object_cols)



from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error

# Function for comparing different approaches
def score_dataset(X_train, X_valid, y_train, y_valid):
    model = RandomForestRegressor(n_estimators=100, random_state=0)
    model.fit(X_train, y_train)
    preds = model.predict(X_valid)
    return mean_absolute_error(y_valid, preds)
```

```
Categorical variables:
['MSZoning', 'Street', 'LotShape', 'LandContour', 'Utilities', 'LotConfig', 'LandSlope', 'Condition1', 'Condition2',
'BldgType', 'HouseStyle', 'RoofStyle', 'RoofMatl', 'ExterQual', 'ExterCond', 'Foundation', 'Heating', 'HeatingQC', 'C
entralAir', 'KitchenQual', 'Functional', 'PavedDrive', 'SaleType', 'SaleCondition']
```

In [7]:
```python
# Score from Approach 1 (Drop Categorical Variables)

drop_X_train = X_train.select_dtypes(exclude=['object'])
drop_X_valid = X_valid.select_dtypes(exclude=['object'])

print("MAE from Approach 1 (Drop categorical variables):")
print(score_dataset(drop_X_train, drop_X_valid, y_train, y_valid))
```

```
MAE from Approach 1 (Drop categorical variables):
17952.591404109586
```

```python
# Score from Approach 2 (Ordinary Encoding)

from sklearn.preprocessing import OrdinalEncoder

# Make copy to avoid changing original data
label_X_train = X_train.copy()
label_X_valid = X_valid.copy()

# Apply ordinal encoder to each column with categorical data
ordinal_encoder = OrdinalEncoder()
label_X_train[object_cols] = ordinal_encoder.fit_transform(X_train[object_cols])
label_X_valid[object_cols] = ordinal_encoder.transform(X_valid[object_cols])

print("MAE from Approach 2 (Ordinal Encoding):")
print(score_dataset(label_X_train, label_X_valid, y_train, y_valid))
```

In [9]:
```python
from sklearn.preprocessing import OneHotEncoder

# Apply one-hot encoder to each column with categorical data
OH_encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)
OH_cols_train = pd.DataFrame(OH_encoder.fit_transform(X_train[object_cols]))
OH_cols_valid = pd.DataFrame(OH_encoder.transform(X_valid[object_cols]))

# One-hot encoding removed index; put it back
OH_cols_train.index = X_train.index
OH_cols_valid.index = X_valid.index

# Remove categorical columns (will replace with one-hot encoding)
num_X_train = X_train.drop(object_cols, axis=1)
num_X_valid = X_valid.drop(object_cols, axis=1)

# Add one-hot encoded columns to numerical features
OH_X_train = pd.concat([num_X_train, OH_cols_train], axis=1)
OH_X_valid = pd.concat([num_X_valid, OH_cols_valid], axis=1)

print("MAE from Approach 3 (One-Hot Encoding):")
print(score_dataset(OH_X_train, OH_X_valid, y_train, y_valid))
```

```
MAE from Approach 3 (One-Hot Encoding):
17514.224246575344
```

5. Pipelines

1. Introduction :
   Pipelines are a simple way to keep your data preprocessing and modeling code organized. Specifically, a pipeline bundles preprocessing and modeling steps so you can use the whole bundle as if it were a single step. Many data scientists hack together models without pipelines, but piplines have some important benefits.

   - Clearner code : Accounting for data at each step of preprocessing can get messy. Wiht a pipliner, you won't need to manually keep track of your training and validation data at each top.
   - Fewer Bugs : There are fewer opportunities to misapply a step or forget a preprocessing step.
   - Easier to Productionize : It can be surprisingly hard to transition a model from a prototype to something deployable at scale. We won't go into the many related concerns here, but piplines can help.
   - More Options for Model Validation : You will see an example in the next tutorial, which cover cross-validation.

In [11]:
```python
# Progress of Machine Learning

# Preprocessing

import pandas as pd
from sklearn.model_selection import train_test_split

# Read the data

data = pd.read_csv('../../KAGGLE/Kaggle_House_Price/train.csv')

# Separate target from predictors

y = data.SalePrice
X = data.drop(['SalePrice'], axis = 1)

# Divide data into training and validation datasets

X_train_full, X_valid_full, y_train, y_valid = train_test_split(X, y, train_size = 0.8, test_size = 0.2, random_stat
e = 0)


# "Cardinality" means the number of unique values in a column
# Select categorical columns with relatively low cardinality (convenient but arbitrary)

categorical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].nunique() < 10 and X_train_full[c
name].dtype == 'object']

# Select numerical columns

numerical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].dtype in ['int64', 'float64']]

# Keep selected columns only

my_cols = categorical_cols + numerical_cols
X_train = X_train_full[my_cols].copy()
X_valid = X_valid_full[my_cols].copy()
```

Step 1 : Define Preprocesisng Steps
Similar to how a pipeline bundles together preprocessing and modeling steps, we use the ColumnTransformer class to bundle together different preprocessing steps.

In [13]:
```python
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# Preprocessing for numerical data

numerical_transformer = SimpleImputer(strategy = 'constant')

# Preprocessing for categorical data

categorical_transformer = Pipeline(steps = [
    ('imputer', SimpleImputer(strategy = 'most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown = 'ignore'))
])

# Bundle preprocessing for numercal and categorical data

preprocessor = ColumnTransformer(
    transformers = [
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])
```

Step2 : Define the Model
Next, we define a random forest model with the familiar RandomForestRegressor class.

In [15]:
```python
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators = 100, random_state = 0)
```

Step3 : Create and Evaluate the Pipeline
Finally, we use the Pipeline class to define a pipeline that bundles the prerprocessing and modeling steps. There are few important things to notice :

- With the pipeline, we preprocess the training data and fit the model in a single line of code.
- With the pipeline, we supply the unprocessed features in X_valid to the predict command, and the pipeline automatically preprocesses the features before generating predictions.

In [17]:
```python
from sklearn.metrics import mean_absolute_error

# Bundle preprocessing and modeling code in a pipeline

my_pipeline = Pipeline(steps = [('preprocessor', preprocessor),
                                ('model', model)])

# Preprocessing of training data, fit model

my_pipeline.fit(X_train, y_train)

# Preprocessing of validation data, get predcitions

preds = my_pipeline.predict(X_valid)

# Evaluate the model

score = mean_absolute_error(y_valid, preds)
print("MAE : ", score)
```

```
MAE :  17740.290308219177
```

In [ ]:
```python
# Final process for prediction of test data

preds_test = my_pipeline.predict(X_test)

output = pd.DataFrame({'Id': X_test.index,
                       'SalePrice': preds_test})
output.to_csv('submission.csv', index=False)
```

6. Cross-Validation

1. Introduction :

Machine learning is an iterative process.

You will face choices about what predictive variables to use, what types of models to use, what arguments to supply to those modles, etc. So far, you have made these choices in a data-driven way by measuring model quality with a validation set.

But there are some drawbacks to this approach. To see this, imagine you have a dataset with 5000 rows. You will typically keep about 20% of the data as a validation dataset, or 1000 rows. But this leaves some random chance in determining model score. That is, a model might do well on one set of 1000 rows, even if it would be inaccurate on a different 1000 rows.

At an extreme, you could imagine having only 1 row of data in the validation set. If you compare alternative models, which one makes the best predictions on a single data point will be mostly a matter of luck!

In general, the larget the validation set, the less randomness there is in our measure of model quality, and the more reliable it will be. Unfortunately, we can only get a large validation set by removing rows from our training data, and smaller training datasets mean worse models.

1. What is cross-validation? :

In cross-validation, we run our modeling process on different subsets of the data to get multiple measures of model quality.

For example, we could begin by dividing the data into 5 pieces, each 20% of the full datset. In this case, we say that we have broken the data into 5 "folds". Then, we run one experiment for each fold

Experiment1	Validation	Training	Training	Training	Training
Ex2		Validation			
Ex3			Validation		
...					

- In Experiment 1, we use the first fold as a validation set and everything else as training data. THis gives us a measure of model quality based on a 20% holdout set.
- In Experiment 2, we hold out data from the second fold. The holdout set is then used to get ad second estimate of model quality.
- We repeat this process, using every fold once as the holdout set. Putting this together, 100% of the daa is used as holdout at some point, and we end up with a measure of model quality that is based on all of the rows in the dataset.

1. When should you use cross-validation ? :

Cross-validation gives a more accurate measure of model quality, which is especially important if you are making a lot of modeling decisions. However, it can take longer to run, because it estimates multiple models.

So, given these tradeoffs, when should you use each approach?

- For small datasets, where extra computational burden isn't a big deal, you should run cross-validation
- For larger datasets, a single validation set is sufficient. Your code will run faster, and you may have enough data that there's little need to re-use some of it for holdout.

There's no simple threshold for what consistutes a large vs. small dataset. But if your model takes a couple minutes or less to run, it's probably worth switching to cross-validation. Alternatively, you can run cross-validation and see if the scores for each expreiment seem close. If each experiment yields the same result, a single validation set is probably sufficient.

In [18]:
```python
# Progress of Machine Learning

# Preprocessing

import pandas as pd
from sklearn.model_selection import train_test_split

# Read the data

data = pd.read_csv('../../KAGGLE/Kaggle_House_Price/train.csv')

# Separate target from predictors

y = data.SalePrice
X = data.drop(['SalePrice'], axis = 1)
```

In [21]:
```python
# Preprocessing for numerical data

numerical_transformer = SimpleImputer(strategy = 'constant')

# Preprocessing for categorical data

categorical_transformer = Pipeline(steps = [
    ('imputer', SimpleImputer(strategy = 'most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown = 'ignore'))
])

# Bundle preprocessing for numercal and categorical data

preprocessor = ColumnTransformer(
    transformers = [
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

# Built pipeline

my_pipeline = Pipeline(steps = [('preprocessor', preprocessor),
                                 ('model', RandomForestRegressor(n_estimators = 100, random_state = 0))])
```

In [23]:
```python
from sklearn.model_selection import cross_val_score

# Multiply by -1 since sklearn calculates negative MAE

scores = -1 * cross_val_score(my_pipeline, X, y, cv = 5, scoring = 'neg_mean_absolute_error')

print("MAE scores : \n", scores)
```

```
MAE scores :
 [17754.00712329 17653.89431507 17771.54866438 16398.27243151
 19406.79760274]
```

The scoring parameter chooses a measure of model qaulity to report : in this case, we chose negative mean absolute error.

It is a little surpising that we specify negative MAE. Scikit-learn has a convetion where all metrics are difined so a high number is better. Using negatives here allows them to be consistent with that convention, though negative MAE is almost unheard of elsewhere.

In [24]:
```python
print("Average MAE score (across experments) : ")
print(scores.mean())
```

```
Average MAE score (across experments) :
17796.90402739726
```

In [ ]:
```python
# Grid Search with personal function using

def get_score(n_estimators):
    """Return the average MAE over 3 CV folds of random forest model.

    Keyword argument:
    n_estimators -- the number of trees in the forest
    """
    # Replace this body with your own code
    my_pipeline = Pipeline(steps=[
    ('preprocessor', SimpleImputer()),
    ('model', RandomForestRegressor(n_estimators= n_estimators, random_state=0))
    ])

    scores = -1 * cross_val_score(my_pipeline, X, y, cv = 3, scoring = 'neg_mean_absolute_error')
    return scores.mean()

results = {estimator : get_score(estimator) for estimator in np.arange(50, 450, 50)}
```

최종 Roadmap

1. 데이터 셋 로딩
2. EDA
3. Visualization
4. Preprocessing
   - Missing values : Imputer vs 직접 작업
   - Categorical values : OneHotEncoder() vs pd.get_dummies()
5. Make pipeline
   - preprocessing -> model 설정까지 한번에
6. Evaluation
   - cross-validation score
   - GridSearchCV()를 통해 최적의 parameter 찾기

7. XGBoost

1. Introduction :

For much of this course, you have made predictions with the random forest method, which achieves better performance than a single decision tree simply by averaging the predictions of many decision trees.

We refer to the random forest method as an 'ensemble method'. By definition, ensemble methods combine the predictions of several models.

1. What is ensemble ?

http://www.dinnopartners.com/__trashed-4/

1. Gradient Boosting :

Gradient boosting is a method that goes through cycles to iteratively add models into an ensemble.

It begins by initializing the ensemble with a single model, whose predictions can be pretty naive. (Even if its predictions are widly inaccurate, subsequent additions to the ensemble will address those errors.)

Then, we start the cycle :

- First, we use the current ensemble to generate predictions for each observation in the dataset. To make a prediction, we add the predictions from all models in the ensemble.
- These predictions are used to calculate a loss function
- Then, we use the loss function to fit a new model that will be added to the ensemble. Specifically, we determine model parameters so that adding this new model to the ensemble will reduce the loss.
- Finally, we add the new model to ensemble, and ...
- repeat them all!

1. Example

In this example, you'll work with the XGboost library. XGBoost stands for extreme gradient boosting, which is an implementation of gradient boosting with several additional features focused on performance and speed.

```python
from xgboost import XGBRegressor

my_model = XGBRegressor()
my_model.fit(X_train, y_train)

from sklearn.metrics import mean_absolute_error

predictions = my_model.predict(X_valid)
print("Mean Absolute Error: " + str(mean_absolute_error(predictions, y_valid)))
```

Parmeter Tuning
XGBoost has a few parameters that can dramatically affect accuracy and training speed. The first parameters you should understand are :

1. n_estimators :

n_estimators specifies how many times to go through the modeling cycle described above. It is equal to the number of modles that we include in the ensemble.

- Too low a value causes underfitting, which leads to inaccurate predictions on both training data and test data.
- Too high a value causes overfitting, which causes accurate predictions on training data, but inaccurate predictions on test data.

1. early_stoppin_rounds :

early_stopping_rounds offers a way to automatically find the ideal value for n_estimators. Early stopping causes the model to stop iterating when the validation score stops imporiving, 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 iterting

Since random chance sometimes cuases a single round where validation scores don't imporve, you need to specify a number for how many rounds of straight deterioration to allow before stopping. Setting early_stopping_rounds = 5 is a reasonable choice. In this case, we stop after 5 straight rounds of deteriorating validation score.

1. learning_rate :

Instead of getting predictions by simply adding up the predictions from each component model, we can 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. So, we can set a higher value for n_estimators without overfitting. If we use early stopping, the appropriate4 number of trees will be determined automatically.

In general, a small learning rate an dlarge number of estimators will yield more accurate XGBoost models, thought it will also take the model longer to train since it does more iterations through the cycle. As default, XGBoost sets learning_rate = 0.1.

```python
my_model = XGBRegressor(n_estimators=1000, learning_rate=0.05)
my_model.fit(X_train, y_train,
             early_stopping_rounds=5,
             eval_set=[(X_valid, y_valid)],
             verbose=False)
```

1. n_jobs :

On larget datasets where runtime is a consideration, you can use parallelism to build your model fatser. 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 nothig but a distraction. But, it's useful in large datasets where you would otherwise spend a long time wating during the fit command.

```python
my_model = XGBRegressor(n_estimators=1000, learning_rate=0.05, n_jobs=4)
my_model.fit(X_train, y_train,
             early_stopping_rounds=5,
             eval_set=[(X_valid, y_valid)],
             verbose=False)
```

8. Data Leakage

1. Introduction :

Data leakage (or leakage) happens when your training data contains information about the target, but similar data will not be available when the model is used for prediction. This leads to high performance on the training set (and possibly even the validation data), but the model will perform poorly in production.

In other words, leakage causes a model to look accurate until you start making decisions with the model, and then the models becomes very inaccurate. There are two main types of leakages : target leakage and train-test contamination.

1. Target leakage :

Target leakage occurs when your predictors include data that will not be available at the time you make predictions. It is important to think about target leakage in terms of the timing or chronocological order that data becomes available, not merely whether a feature helps make good predictions.

An example will be helpful. Imagine you wnat to predict who wll get sick with pneumonia. THe top few rows of you raw data look like this :

got_pneumonia	age	weight	male	took_antibiotic_medicine	...
False	65	100	False	False	...
False	72	130	True	False	...
True	58	100	False	True	...

People take anitibiotic medicinces after getting pneumonia in order to recover. THe raw data shows a strong relationship between those columns, but took_antibiotic_medicine is frequently changed after the valeu for got_pnumonia is determined.

The model would see that anyone who has a value of Flase for took_antibiotic_medicine didn't have pneumonia. Since avlidation data comes from the same source as training data, the pattern will repeat itself in validation, and the model will have great validation score.

But the model will be very inaccurate hwen subsequently deployed in the real world, because even patients who will get pneunomia won't have received antibiotics yet when we need to make predictions about their future help.

To preven this type of data leakage, any variabel updated after the target value is realized should be excluded.

타겟 변수 이후에 영향을 주는 변수는 필요없음

https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=tjdudwo93&logNo=221085844907

1. Train-Test contamination :

이 data leakage는 학습 데이터와 validation 데이터를 제대로 구분하지 않았을 때 생긴다.

Recall that validation는 모델이 이전에 고려되지 않았던 데이터에 대해 어떻게 작동하는지 측정하는 것을 말한다. Validation 데이터가 전처리에 영향을 준다면 이 과정에 손상이 올 수도 있다.

만약 train_test_split()을 전처리 과정(missing value를 처리하는 imputer 같은) 이전에 한다고 생각해보자. 결과는? Validation 스코어는 좋겠지만 배포 후의 성능은 별로일 것이다.

만약 validation 데이터가 train-test split을 기반으로 만들어졌을 때, validation 데이터를 모든 fitting에서 제외하고, 전처리 단계의 fitting에 포함시켜야 한다. Scikit-learn의 pipelines을 이용하면 더 쉽다. Cross-validation을 사용할 때는 파이프 라인 내에서 전처리를 수행하는 것이 훨씬 더 중요하다.

데이터 전처리는 train-test split과정 이후 train에서만 적용되어야 함. Cross-validation의 경우에는 Pipeline을 생성후 전처리를 통해 수행하는게 정확함.

https://m.blog.naver.com/hongjg3229/221811766581

최종 Roadmap

1. 데이터 셋 로딩
2. EDA
3. Visualization
4. Preprocessing
   - Pipeline을 통해 작업하지 않는경우 train_test_spliter를 통해 데이터를 분류하여 작업하는 것이 중요함.
   - Missing values : Imputer vs 직접 작업
   - Categorical values : OneHotEncoder() vs pd.get_dummies()
5. Make pipeline
   - preprocessing -> model 설정까지 한번에
6. Evaluation
   - cross-validation score
   - GridSearchCV()를 통해 최적의 parameter 찾기