W4995 Applied Machine Learning
Fall 2021

Lecture 4

Dr. Vijay Pappu

Announcements

- Project deliverable 1 due today (10/06) before 11:59 PM EST
- Assignment 2 will be released tomorrow (10/07) and due on 10/20
- Assignment 2 worth 20% of the final grade
- Midterm would include Lectures 1-5

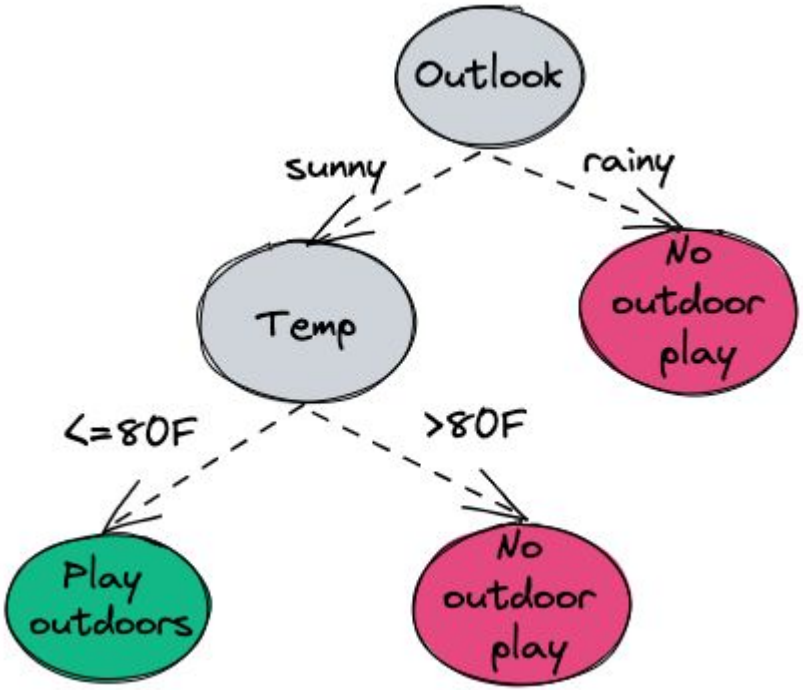
In today’s lecture, we will cover…

- Decision Trees, Ensembles & Bagging
- Random Forests & Gradient Boosting

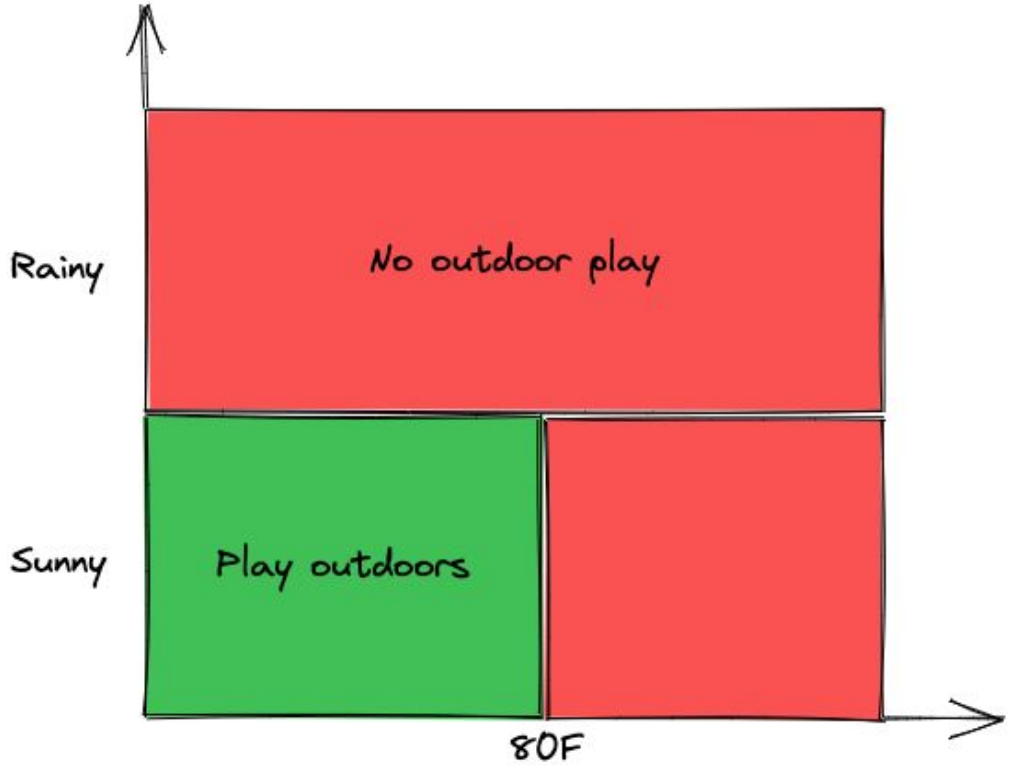
Decision Trees

Decision Trees

- Greedy algorithm
- Applicable to both classification & regression problems
- Easy to interpret & deploy
- Non-linear decision boundary
- Minimal preprocessing
- Invariant to scale of data

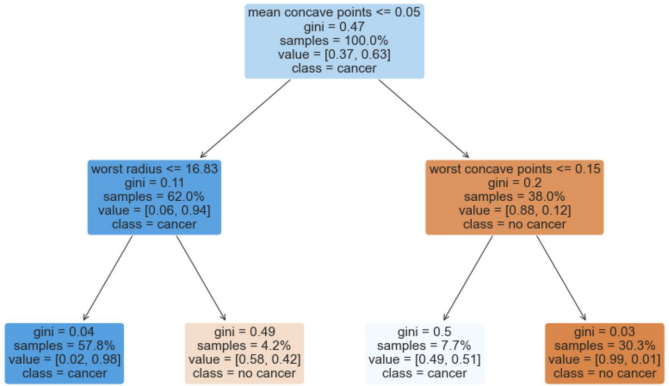


Decision Trees

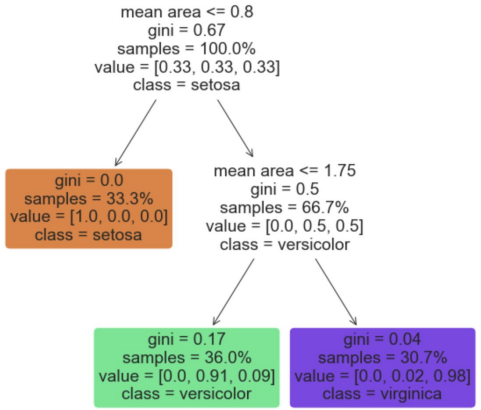


Decision Trees

Classification Trees

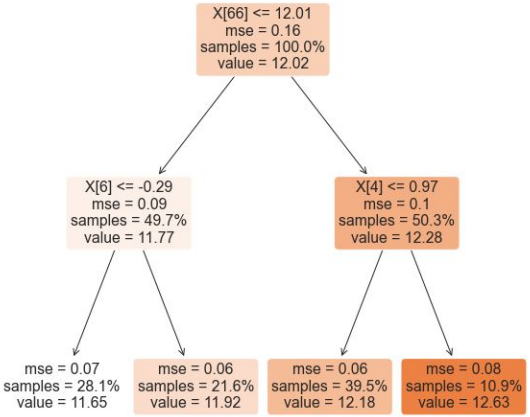


Binary classification



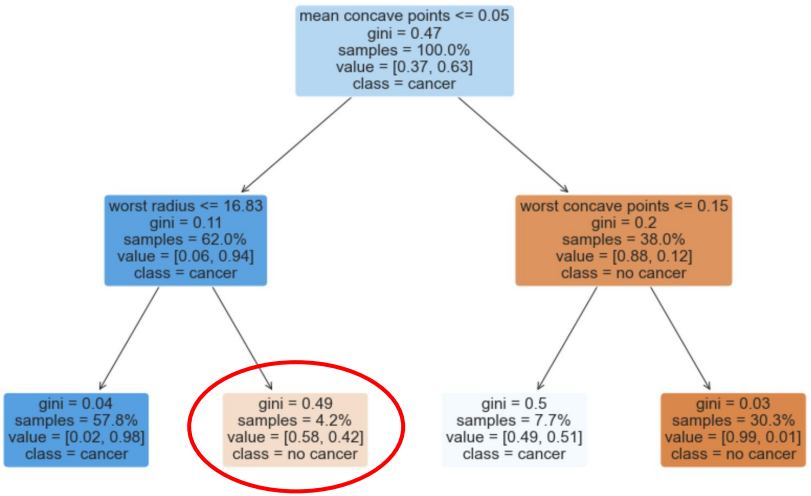
Multi-class classification

Regression Trees



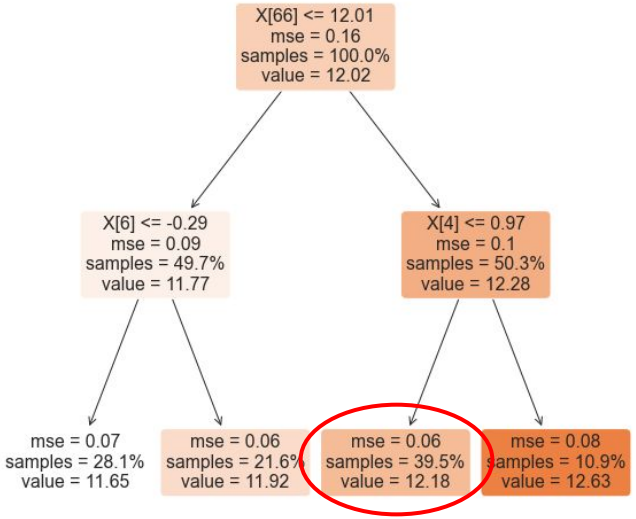
Decision Tree Prediction

Classification Trees



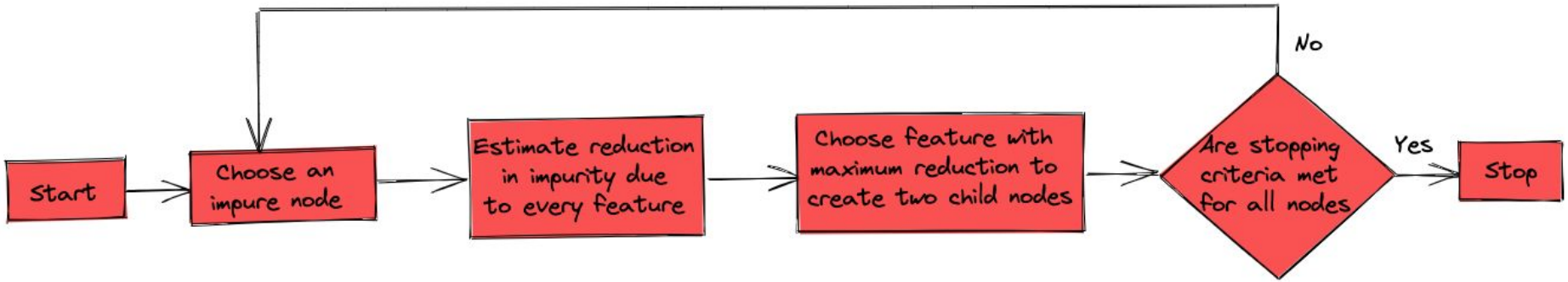
majority voting

Regression Trees

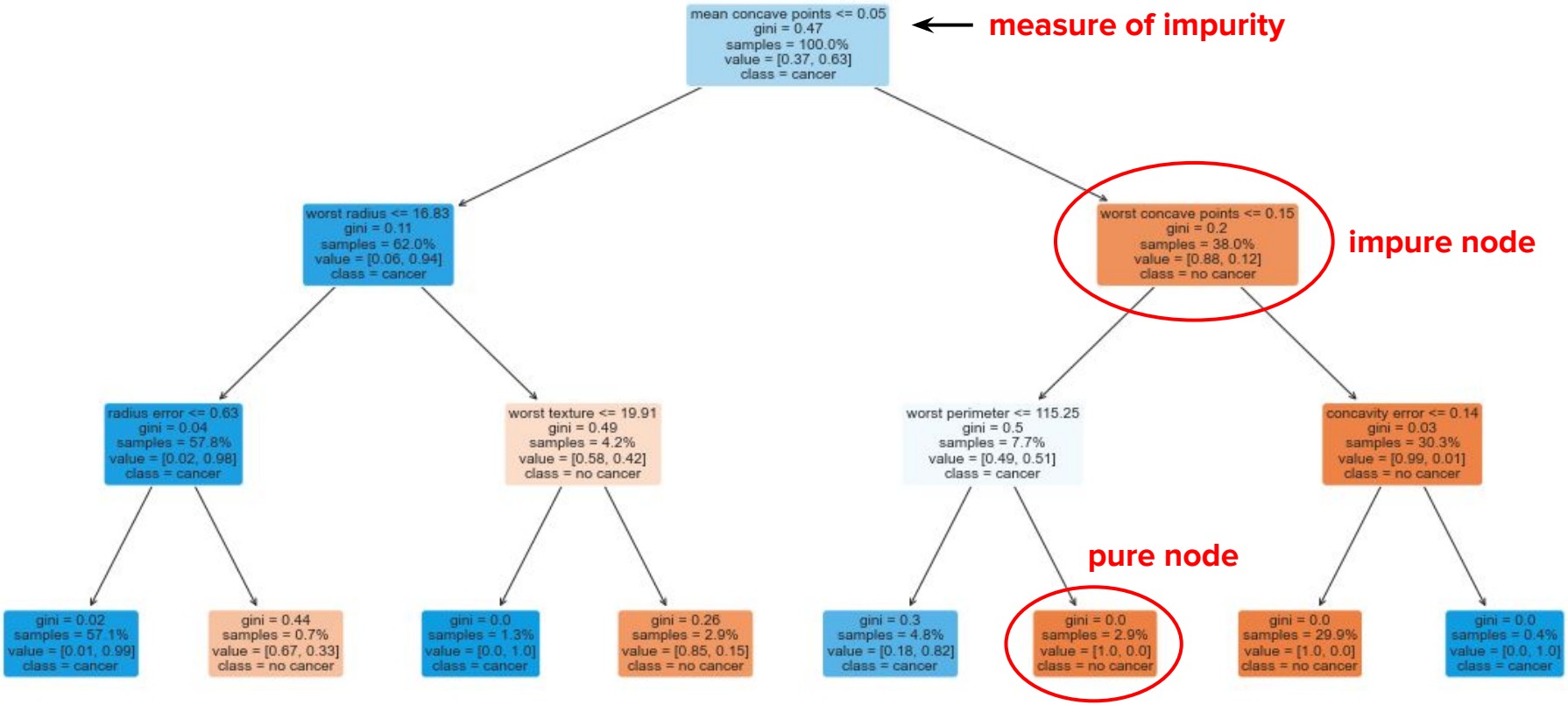


sample mean

Decision Trees



Decision Trees



Classification Trees - Measure of Impurity

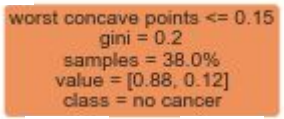
$$Entropy(\ node) \ = \ - \sum_{i=1}^{K} p_{i} \log_{2} p_{i}$$

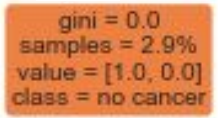
$$Gini\ Index(\ node) \ = 1 - \sum_{i=1}^{K} p_{i}^{2}$$

Classification Trees - Measure of Impurity

$$Entropy(\ node) \ = \ -\ \sum_{i=1}^{K} p_{\ i} \log_2 p_{\ i}$$

$$Gini\ Index(\ node) \ = 1 - \ \sum_{i=1}^{K} p_{\ i}{}^2$$





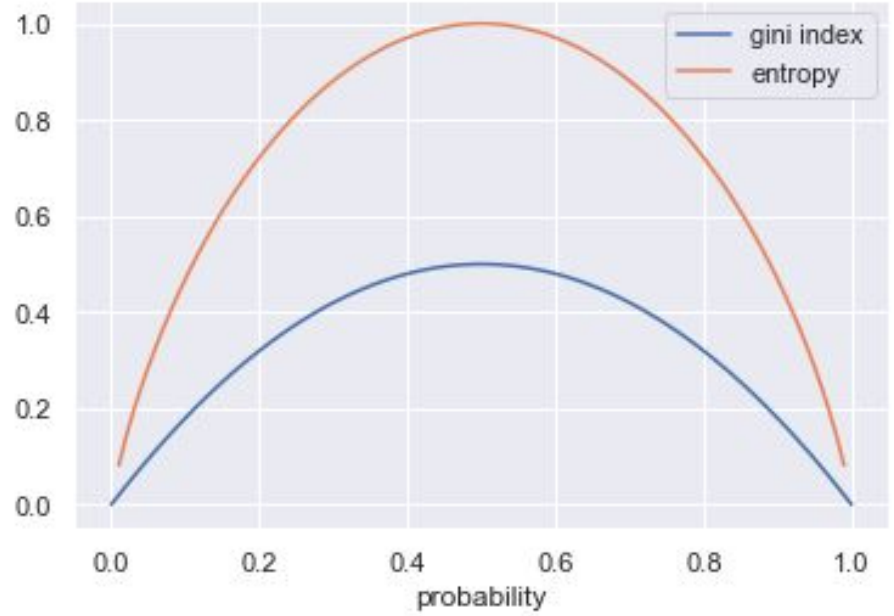
Gini index = 1 - (0.88^2 + 0.12^2) = 0.2

Entropy = -0.88*$\log_2$(0.88) - 0.12*$\log_2$(0.12) = 0.53

Gini index = 1 - (1^2 + 0^2) = 0

Entropy = -1*$\log_2$(1) - 0*$\log_2$(0) = 0

Gini Index v.s. Entropy



Classification Trees - Information Gain

$$S_a(v) = \{x \in T \mid x_a = v\}$$

Conditional Shannon Entropy

$$H(T|a) = \sum_{v \in vals(a)} \frac{|S_a(v)|}{|T|} H(S_a(v))$$

Information Gain

$$IG(T, a) = H(T) - H(T|a)$$

Classification Trees - Information Gain

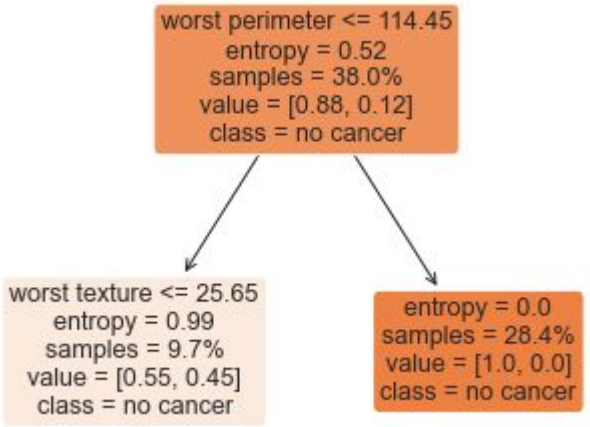
$$S_a(v) = \{x \in T \mid x_a = v\}$$

Conditional Shannon Entropy

$$H(T|a) = \sum_{v \in vals(a)} \frac{|S_a(v)|}{|T|} H(S_a(v))$$

Information Gain

$$IG(T, a) = H(T) - H(T|a)$$



H(T|a) = (9.7/38)*0.99 + (28.4/38)*0 = **0.25**

IG(T, a) = H(T) - H(T|a) = 0.52 - 0.25 = **0.27**

Regression Trees - Measure of Impurity

$$Mean\ squared\ error\left(node_j\right) = \frac{1}{\left|N_j\right|}\sum_{i \in N_j}\left(x_i - \overline{x_j}\right)^2$$

$$Mean\ absolute\ error\left(node_j\right) = \frac{1}{\left|N_j\right|}\sum_{i \in N_j}\left|x_i - \overline{x_j}\right|$$

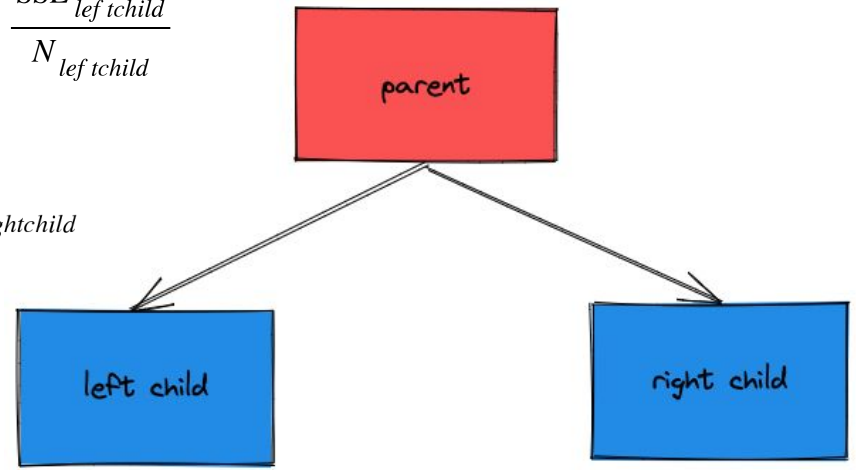
Regression Trees - Sum of Squared Error (SSE)

- An exhaustive search across all categorical features and their categories to find the (feature, subsets) combination with the highest information gain (IG).

$$MSE_{parent} = \frac{SSE_{parent}}{N_{parent}} \qquad MSE_{rightchild} = \frac{SSE_{rightchild}}{N_{rightchild}} \qquad MSE_{leftchild} = \frac{SSE_{leftchild}}{N_{leftchild}}$$

$$Gain = MSE_{parent} - \left(\frac{N_{leftchild}}{N_{parent}}\right) MSE_{leftchild} - \left(\frac{N_{rightchild}}{N_{parent}}\right) MSE_{rightchild}$$

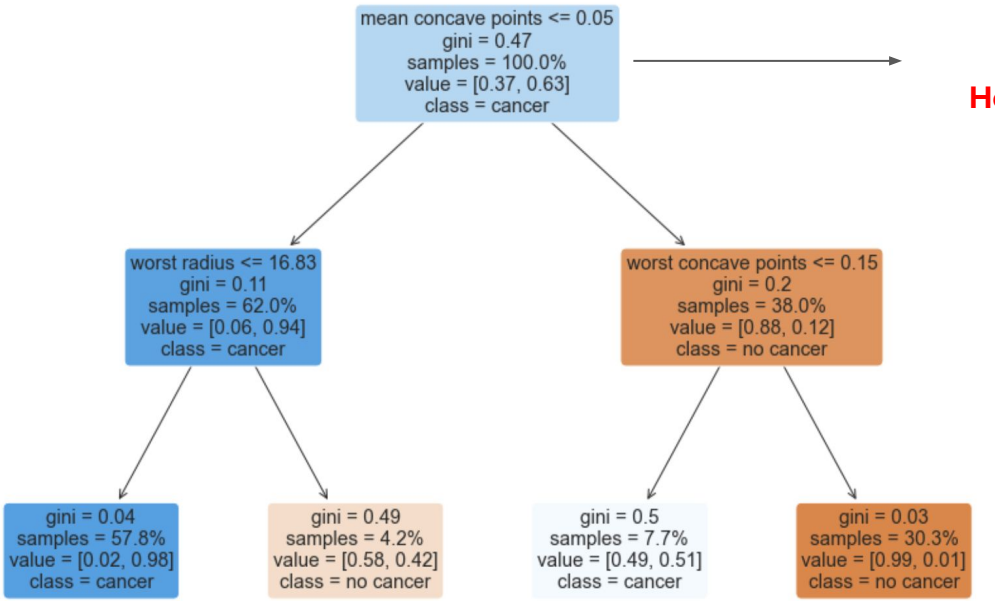
$$Gain = \frac{1}{N_{parent}} \left(SSE_{parent} - SSE_{leftchild} - SSE_{rightchild} \right)$$



Classification Trees v.s. Regression Trees

Classification Trees	Regression Trees
• Applicable to categorical outcomes (binary & multi-class) • Uses entropy or gini index as measure of impurity • Uses information gain to determine the optimal split • Prediction by majority voting	• Applicable to real-valued outcomes • Uses variance/MSE as measure of impurity • Uses Sum of Squared Error (SSE) to determine optimal split • Prediction by sample mean

Decision Trees



How do we find feature to split on?

How do we decide on the threshold?

Node splitting for continuous features

- An exhaustive search across all features and values to find the (feature, threshold) combination with the highest information gain (IG).

```
max_IG = 0
feature_to_split = None
feature_threshold = None
for feature in features:
     for value in feature.values:
          IG(target, feature) = H(target) - H(target | feature, value)
          if IG(target, feature) >= max_IG:
               feature_threshold = value
               feature_to_split = feature
return (feature_to_split, feature_threshold)
```

Node splitting for categorical features

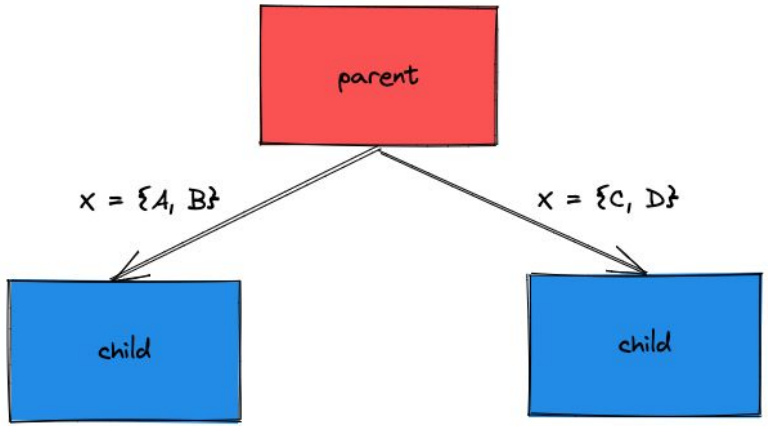
- An exhaustive search across all categorical features and their categories to find the (feature, subsets) combination with the highest information gain (IG).

$X \in \{A, B, C, D\}$

L categories

{A}, {B, C, D}
{B}, {A, C, D}
{C}, {A, B, D}
{D}, {A, B, C}
{A, B}, {C, D}
{A, C}, {B, D}
{A, D}, {B, C}
{B, C}, {A, D}
{B, D}, {A, C}
{C, D}, {A, B}

$O(2^L)$



This is computationally expensive especially when we have features with lots of categories

Node splitting for categorical features

- The categories are ordered by mean response rate (target encoding)
- Optimal split can be found among the $O(2^{L})$ partitions by only evaluating **O(L)** splits of the ordered categories.

Feature	Target
A	1
A	1
A	0
B	0
B	0
D	1
D	1
C	0

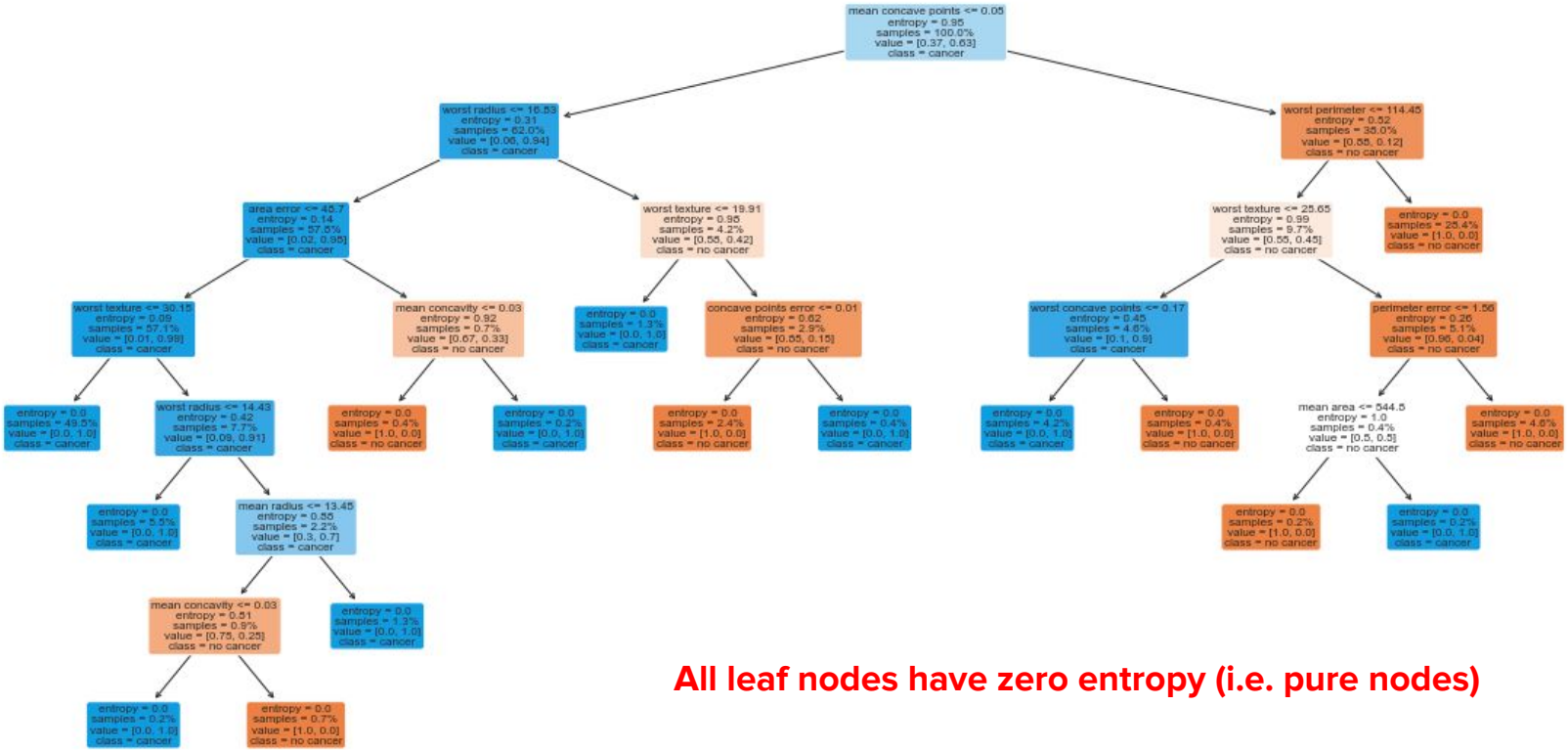
Feature	MRR
D	1
A	0.66
B	0
C	0

{D}, {A, B, C}

{D, A}, {B, C}

{D, A, B}, { C}

Decision Trees - Overfitting



All leaf nodes have zero entropy (i.e. pure nodes)

Decision Trees - Overfitting

```
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()
feature_names = data.feature_names
bc_df = pd.DataFrame(data.data, columns = feature_names)
target = pd.Series(data.target)
print("Class distribution:")
print(target.value_counts())
print("Dataset size:")
print(bc_df.shape)
dev_X, test_X, dev_y, test_y = train_test_split(bc_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), feature_names))
pipe = make_pipeline(preprocess, DecisionTreeClassifier(max_depth=10, criterion="entropy"))
pipe.fit(dev_X, dev_y)
print(pipe.score(dev_X, dev_y))
print(pipe.score(test_X, test_y))
```

```
Class distribution:
1    357
0    212
dtype: int64
Dataset size:
(569, 30)
1.0
0.9473684210526315
```

Decision Trees - Overfitting

```
from sklearn.compose import make_column_transformer
from sklearn.tree import DecisionTreeRegressor

ohe_features = [
    "LotShape",
    "Street",
    "ExterCond",
    "OverallCond",
    "OverallQual",
    "Condition1",
    "Functional",
    "MSZoning",
    "FireplaceQu",
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
    "TotalBsmtSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                remainder="passthrough"
                       )
pipe = make_pipeline(preprocess, DecisionTreeRegressor(max_depth=10))
pipe.fit(dev_X, dev_y)
print(pipe.score(dev_X, dev_y))
print(pipe.score(test_X, test_y))
```

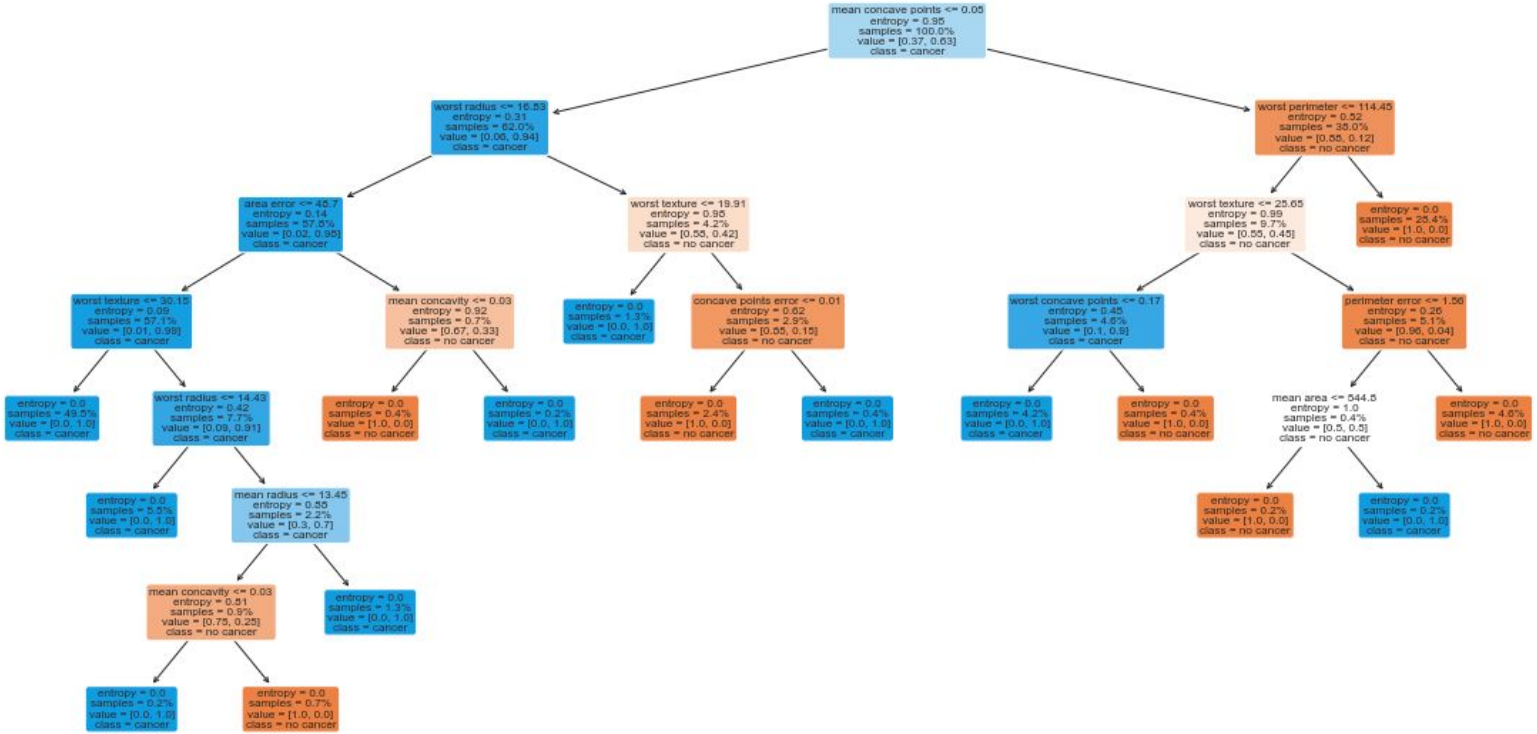
```
0.9747390421323957
0.7615335932514169
```

Decision Trees - Preventing Overfitting

- **Pruning**
  - Reduced error
  - Cost complexity
- **Early stopping**
  - Maximum depth
  - Maximum leaf nodes
  - Minimum samples split
  - Minimum impurity decrease

Decision Trees - Pruning

- The tree is built out completely with all leaf nodes being pure.



Decision Tree Pruning - Reduced Error

- Starting at the leaves, each node is replaced with its most popular class.
- If the validation metric is not negatively affected, then the change is kept, else it is reverted.
- Reduced error pruning has the advantage of speed and simplicity.

Decision Tree Pruning - Cost Complexity

- The cost complexity measure $R_\alpha(T)$ of a tree T is given by:

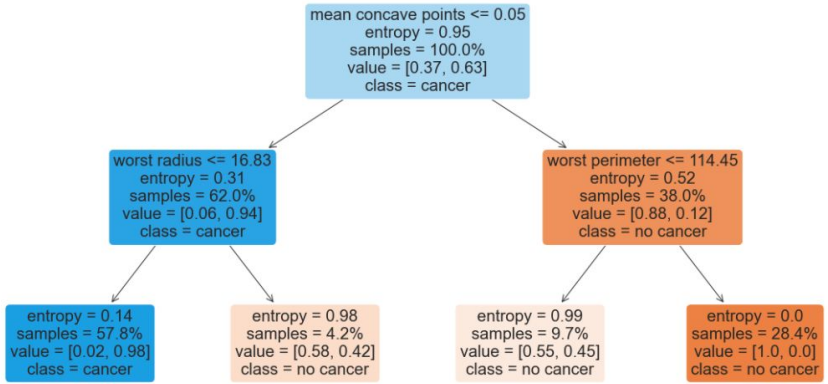
$$R_\alpha(T) = R(T) + \alpha|T|$$

- $R(T_t)$ is defined as sum of impurities for all leaf nodes of a tree rooted at node *t*.

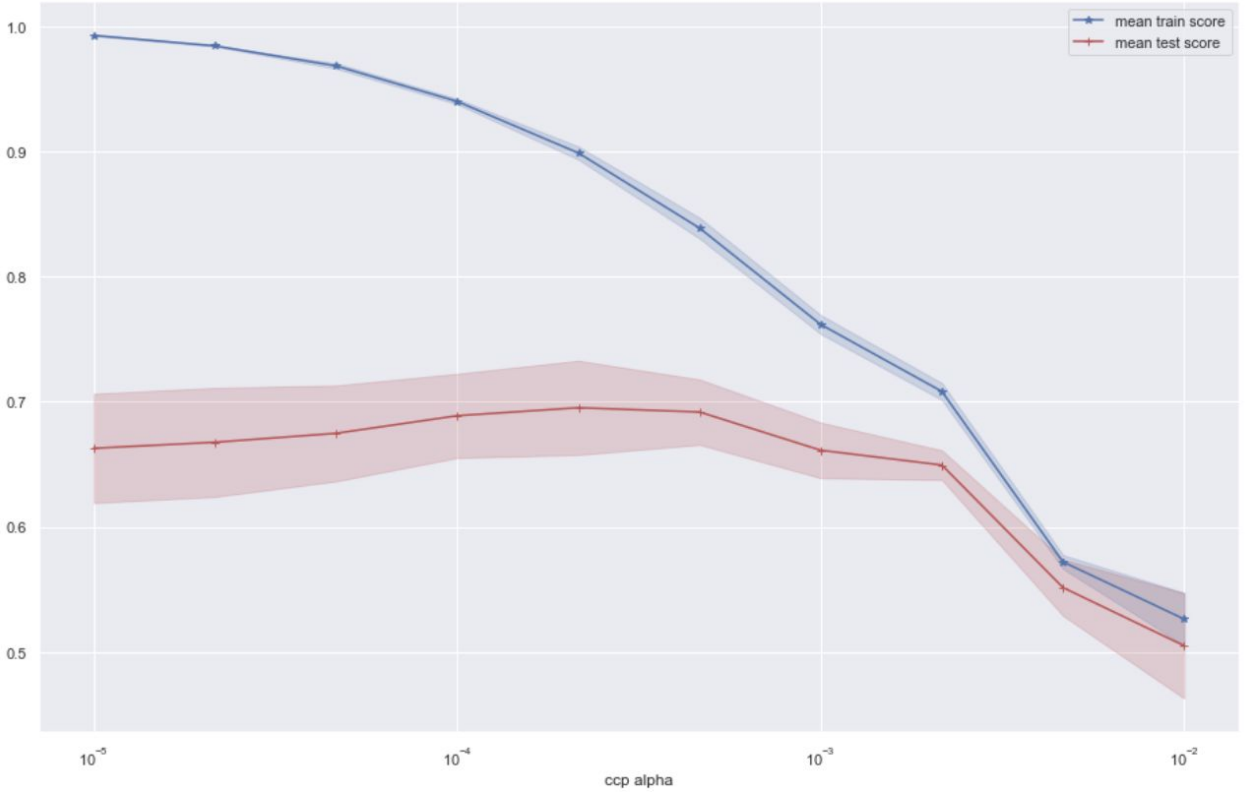
$$R(T_t) = \sum_{i \in leaf\ nodes} R(i)$$

- The node with the least α_{eff} is pruned:

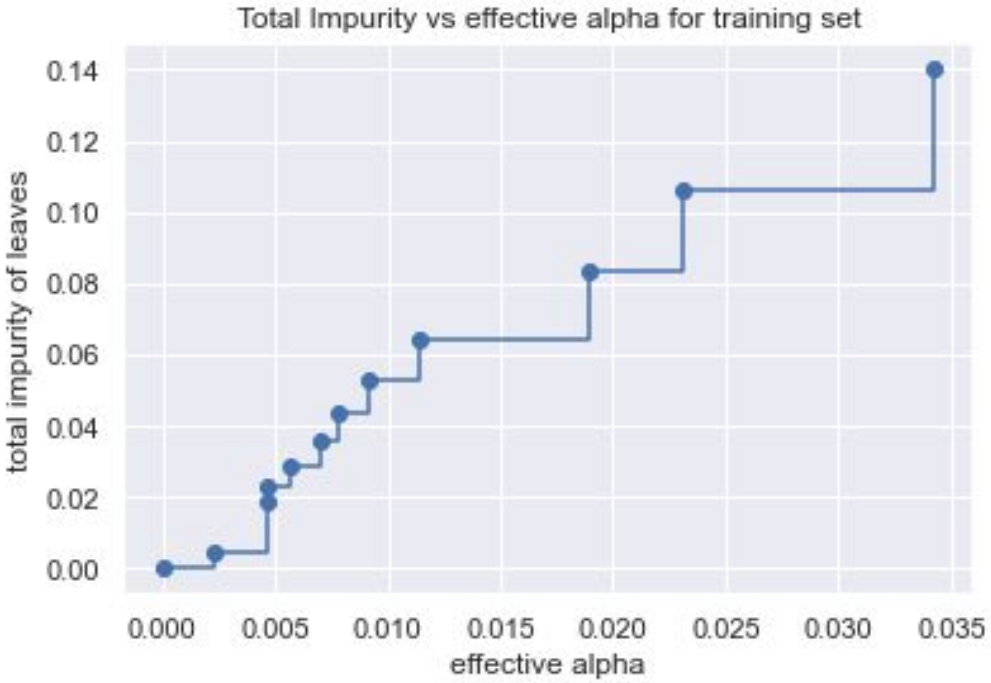
$$\alpha_{eff} = \frac{R(t) - R_\alpha(T_t)}{T - 1}$$



Decision Tree Pruning - Cost Complexity



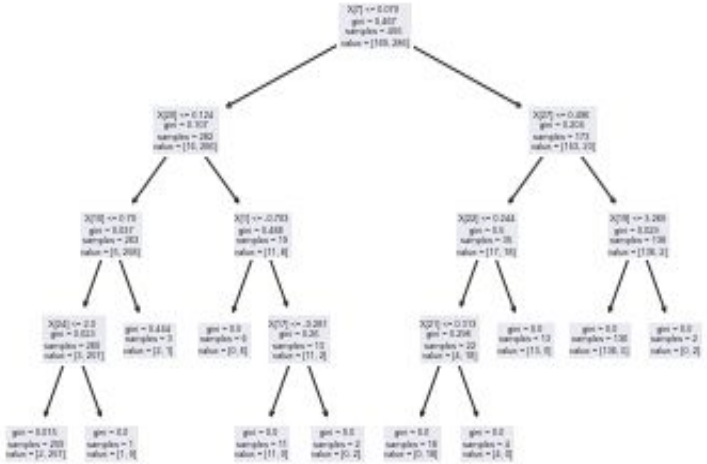
Decision Tree Pruning - Cost Complexity



Decision Tree Pruning - Cost Complexity

```
Best score: 0.9142857142857143
Best alpha: {'max_depth': 10}
Test score: 0.9210526315789473
```

```
Best score: 0.9274725274725275
Best alpha: {'ccp_alpha': 0.00379269019073225}
Test score: 0.9298245614035088
```



Decision Tree Early Stopping - Maximum Depth

```
from sklearn.compose import make_column_transformer
from sklearn.tree import DecisionTreeRegressor

ohe_features = [
    "LotShape", "Street", "ExterCond", "OverallCond",
    "OverallQual", "Condition1", "Functional", "MSZoning",
    "FireplaceQu",
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr", "BsmtFinSF1", "BsmtFullBath", "EnclosedPorch",
    "GarageArea", "LotArea", "1stFlrSF", "2ndFlrSF",
    "TotalBsmtSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                            )
pipe = make_pipeline(preprocess,
                     GridSearchCV(DecisionTreeRegressor(random_state=84),
                                  param_grid = [{"max_depth":np.arange(1, 10)}],
                                  return_train_score=True))
pipe.fit(dev_X, dev_y)
grid_search_results = pipe.named_steps["gridsearchcv"]
print(f"Best score:", grid_search_results.best_score_)
print(f"Best alpha:", grid_search_results.best_params_)
print(f"Test score:", pipe.score(test_X, test_y))
```

```
Best score: 0.6866253931582456
Best alpha: {'max_depth': 6}
Test score: 0.7638713832337403
```

Decision Tree Early Stopping - Maximum Leaf Node

```
from sklearn.compose import make_column_transformer
from sklearn.tree import DecisionTreeRegressor

ohe_features = [
    "LotShape", "Street", "ExterCond", "OverallCond",
    "OverallQual", "Condition1", "Functional", "MSZoning",
    "FireplaceQu",
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr", "BsmtFinSF1", "BsmtFullBath", "EnclosedPorch",
    "GarageArea", "LotArea", "1stFlrSF", "2ndFlrSF",
    "TotalBsmtSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                      (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                      (TargetEncoder(handle_unknown="ignore"), te_features),
                                remainder="passthrough"
                     )
pipe = make_pipeline(preprocess,
                    GridSearchCV(DecisionTreeRegressor(random_state=84),
                                 param_grid = [{"max_leaf_nodes":np.arange(2, 64)}],
                                 return_train_score=True))
pipe.fit(dev_X, dev_y)
grid_search_results = pipe.named_steps["gridsearchcv"]
print(f"Best score:", grid_search_results.best_score_)
print(f"Best alpha:", grid_search_results.best_params_)
print(f"Test score:", pipe.score(test_X, test_y))
```

```
Best score: 0.6978349755853073
Best alpha: {'max_leaf_nodes': 35}
Test score: 0.7597972729316161
```

Decision Tree Early Stopping - Minimum Sample Split

```
from sklearn.compose import make_column_transformer
from sklearn.tree import DecisionTreeRegressor

ohe_features = [
    "LotShape", "Street", "ExterCond", "OverallCond",
    "OverallQual", "Condition1", "Functional", "MSZoning",
    "FireplaceQu",
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr", "BsmtFinSF1", "BsmtFullBath", "EnclosedPorch",
    "GarageArea", "LotArea", "1stFlrSF", "2ndFlrSF",
    "TotalBsmtSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                       )
pipe = make_pipeline(preprocess,
                     GridSearchCV(DecisionTreeRegressor(random_state=84),
                                  param_grid = [{"min_samples_split":np.linspace(0, 1, 21)}],
                                  return_train_score=True))
pipe.fit(dev_X, dev_y)
grid_search_results = pipe.named_steps["gridsearchcv"]
print(f"Best score:", grid_search_results.best_score_)
print(f"Best alpha:", grid_search_results.best_params_)
print(f"Test score:", pipe.score(test_X, test_y))
```

```
Best score: 0.7008977007402442
Best alpha: {'min_samples_split': 0.05}
Test score: 0.7709146523012278
```

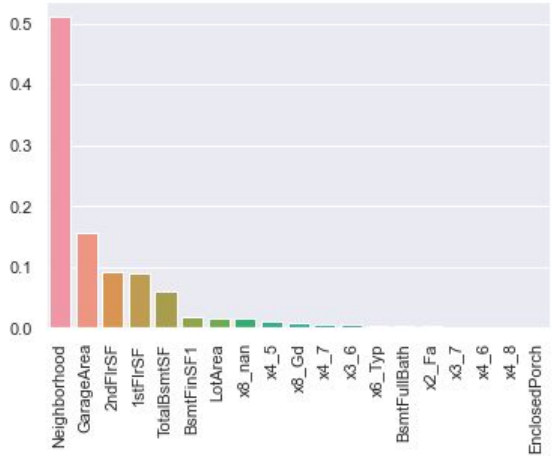
Decision Tree Early Stopping - Minimum Impurity Decrease

```
from sklearn.compose import make_column_transformer
from sklearn.tree import DecisionTreeRegressor

ohe_features = [
    "LotShape", "Street", "ExterCond", "OverallCond",
    "OverallQual", "Condition1", "Functional", "MSZoning",
    "FireplaceQu",
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr", "BsmtFinSF1", "BsmtFullBath", "EnclosedPorch",
    "GarageArea", "LotArea", "1stFlrSF", "2ndFlrSF",
    "TotalBsmtSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                remainder="passthrough"
                       )
pipe = make_pipeline(preprocess,
                     GridSearchCV(DecisionTreeRegressor(random_state=84),
                                  param_grid = [{"min_impurity_decrease":np.logspace(-3, -1, 100)}],
                                  return_train_score=True))
pipe.fit(dev_X, dev_y)
grid_search_results = pipe.named_steps["gridsearchcv"]
print(f"Best score:", grid_search_results.best_score_)
print(f"Best alpha:", grid_search_results.best_params_)
print(f"Test score:", pipe.score(test_X, test_y))
best_tree = grid_search_results.best_estimator_
tree_dot = plot_tree(best_tree)
```

```
Best score: 0.662573889739327
Best alpha: {'min_impurity_decrease': 0.0010974987654930556}
Test score: 0.7230498522893216
```

Decision Tree - Feature Importance

```
ohe_feature_names = preprocess.named_transformers_["onehotencoder"].get_feature_names().tolist()
te_feature_names = preprocess.named_transformers_["targetencoder"].get_feature_names()
feature_names = num_features + ohe_feature_names + te_feature_names
feat_imps = zip(feature_names, best_tree.feature_importances_)
feats, imps = zip(*(sorted(list(filter(lambda x: x[1] != 0, feat_imps)), key=lambda x: x[1], reverse=True)))
ax = sns.barplot(list(feats), list(imps))
ax.tick_params(axis='x', rotation=90)
```

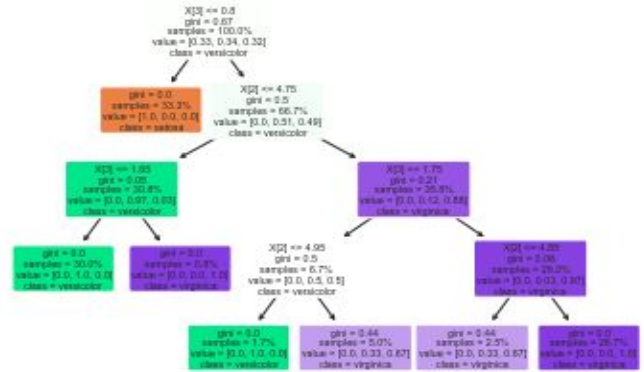


Ensemble methods

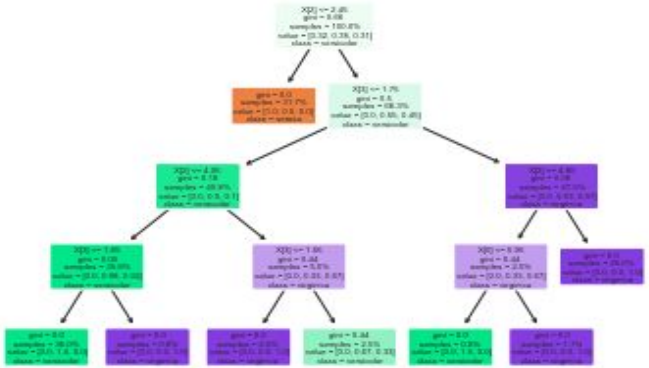
Ensemble methods - Motivation

- The decision trees are highly **unstable** and can structurally change with slight variation in input data.

```
iris = datasets.load_iris()
X = iris.data
y = iris.target
dev_X, test_X, dev_y, test_y = train_test_split(X, y,
                                                test_size=0.2,
                                                random_state=42)
tree = DecisionTreeClassifier(max_depth= 4)
tree.fit(dev_X, dev_y)
tree_plot = plot_tree(tree,
                      filled=True, proportion=True,
                      rounded=True,
                      class_names =iris.target_names,
                      precision=2)
```

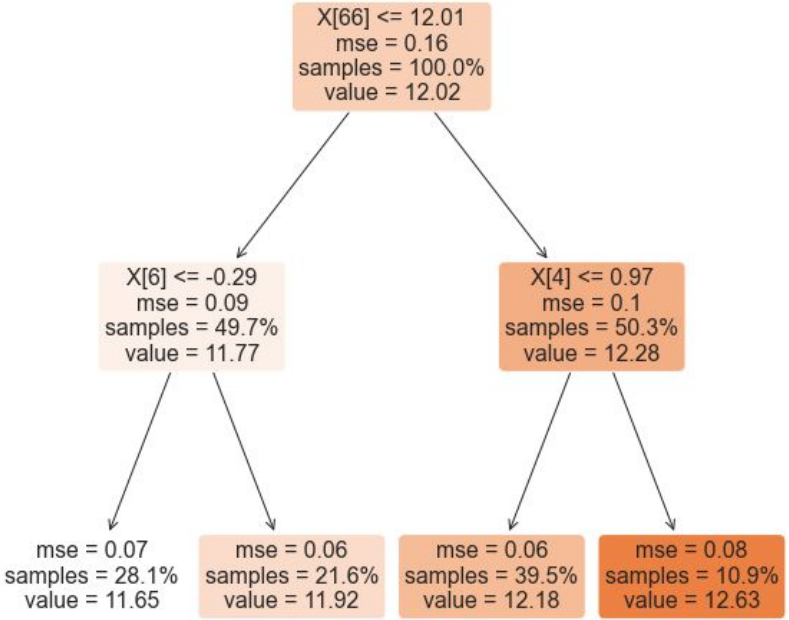


```
iris = datasets.load_iris()
X = iris.data
y = iris.target
dev_X, test_X, dev_y, test_y = train_test_split(X, y,
                                                test_size=0.2,
                                                random_state=84)
tree = DecisionTreeClassifier(max_depth= 4)
tree.fit(dev_X, dev_y)
tree_plot = plot_tree(tree,
                      filled=True, proportion=True,
                      rounded=True,
                      class_names =iris.target_names,
                      precision=2)
```



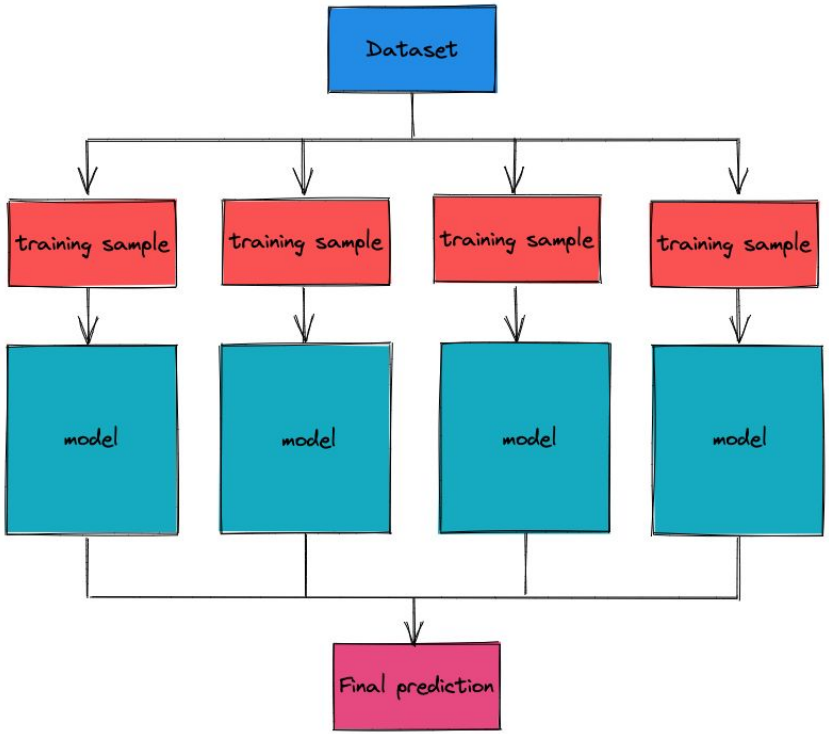
Ensemble methods - Motivation

- Decision trees perform poorly on continuous outcomes (regression) due to ***limited*** model capacity.



Ensemble methods

- Several weak/simple learners are combined to make the final prediction
- Generally ensemble methods aim to reduce model variance.
- Ensemble methods improve performance especially if the individual learners are not correlated.
- Depending on training sample construction and output aggregation, there are two categories:
    - **Bagging (Bootstrap aggregation)**
    - **Boosting**



Simple Ensemble Classifier - Example

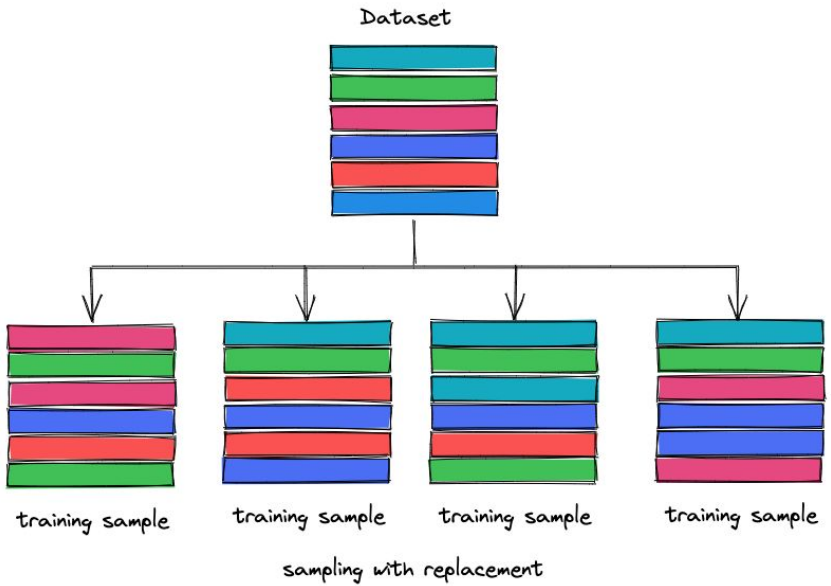
```
from sklearn.compose import make_column_transformer
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import VotingRegressor
from sklearn.linear_model import Ridge

ohe_features = [
    "LotShape", "Street", "ExterCond", "OverallCond",
    "OverallQual", "Condition1", "Functional", "MSZoning",
    "FireplaceQu",
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr", "BsmtFinSF1", "BsmtFullBath", "EnclosedPorch",
    "GarageArea", "LotArea", "1stFlrSF", "2ndFlrSF",
    "TotalBsmtSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)

preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                       )
pipe = make_pipeline(preprocess,
                     VotingRegressor(
                     [('logreg', LinearRegression()), ('tree', DecisionTreeRegressor(max_depth=4))]))
pipe.fit(dev_X, dev_y)
lr, tree = pipe.named_steps['votingregressor'].estimators_
print(f"Decision Tree score:", tree.score(preprocess.transform(test_X), test_y))
print(f"Linear Regression score:", lr.score(preprocess.transform(test_X), test_y))
print(f"Voting Regression score:", pipe.score(test_X, test_y))
```

```
Decision Tree score: 0.7048317456969704
Linear Regression score: 0.9044704028619821
Voting Regression score: 0.8633279883508591
```

Does not always work!

Bagging (Bootstrap aggregation)

- Several training samples (of same size) are created by sampling the dataset with replacement
- Each training sample is then used to train a model
- The outputs from each of the models are averaged to make the final prediction.



Bagging - Example

```
from sklearn.compose import make_column_transformer
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import VotingRegressor, BaggingRegressor
from sklearn.linear_model import Ridge

ohe_features = [
    "LotShape", "Street", "ExterCond", "OverallCond",
    "OverallQual", "Condition1", "Functional", "MSZoning",
    "FireplaceQu",
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr", "BsmtFinSF1", "BsmtFullBath", "EnclosedPorch",
    "GarageArea", "LotArea", "1stFlrSF", "2ndFlrSF",
    "TotalBsmtSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = data_df["SalePrice"]
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)

preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                remainder="passthrough"
                       )
pipe_bagging_tree = make_pipeline(preprocess, BaggingRegressor(DecisionTreeRegressor(), n_estimators=10))
pipe_bagging_tree.fit(dev_X, dev_y)
pipe_bagging_linreg = make_pipeline(preprocess, BaggingRegressor(LinearRegression(), n_estimators=10))
pipe_bagging_linreg.fit(dev_X, dev_y)
pipe_tree = make_pipeline(preprocess, DecisionTreeRegressor())
pipe_tree.fit(dev_X, dev_y)
pipe_linreg = make_pipeline(preprocess, LinearRegression())
pipe_linreg.fit(dev_X, dev_y)

print(f"Decision Regressor score:", pipe_tree.score(test_X, test_y))
print(f"Bagging Tree Regressor score:", pipe_bagging_tree.score(test_X, test_y))
print(f"Linear Regressor score:", pipe_linreg.score(test_X, test_y))
print(f"Bagging Linear Regressor score:", pipe_bagging_linreg.score(test_X, test_y))
```

```
Decision Regressor score: 0.7468813564462373
Bagging Tree Regressor score: 0.8707348849093468
Linear Regressor score: 0.8677511203681992
Bagging Linear Regressor score: 0.8781681829515994
```

Bagging - Example

```
from sklearn.compose import make_column_transformer
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import VotingRegressor, BaggingRegressor
from sklearn.linear_model import Ridge

ohe_features = [
    "LotShape", "Street", "ExterCond", "OverallCond",
    "OverallQual", "Condition1", "Functional", "MSZoning",
    "FireplaceQu",
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr", "BsmtFinSF1", "BsmtFullBath", "EnclosedPorch",
    "GarageArea", "LotArea", "1stFlrSF", "2ndFlrSF",
    "TotalBsmtSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)

preprocess = make_column_transformer((StandardScaler(), num_features),
                         (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                         (TargetEncoder(handle_unknown="ignore"), te_features),
                                  remainder="passthrough"
                        )
pipe_bagging_tree = make_pipeline(preprocess, BaggingRegressor(DecisionTreeRegressor(), n_estimators=10))
pipe_bagging_tree.fit(dev_X, dev_y)
pipe_bagging_linreg = make_pipeline(preprocess, BaggingRegressor(LinearRegression(), n_estimators=10))
pipe_bagging_linreg.fit(dev_X, dev_y)
pipe_tree = make_pipeline(preprocess, DecisionTreeRegressor())
pipe_tree.fit(dev_X, dev_y)
pipe_linreg = make_pipeline(preprocess, LinearRegression())
pipe_linreg.fit(dev_X, dev_y)

print(f"Decision Regressor score:", pipe_tree.score(test_X, test_y))
print(f"Bagging Tree Regressor score:", pipe_bagging_tree.score(test_X, test_y))
print(f"Linear Regressor score:", pipe_linreg.score(test_X, test_y))
print(f"Bagging Linear Regressor score:", pipe_bagging_linreg.score(test_X, test_y))
```

```
Decision Regressor score: 0.727789235296771
Bagging Tree Regressor score: 0.84758567132336
Linear Regressor score: 0.904470402861982l
Bagging Linear Regressor score: 0.881739975304296
```

Again, does not always work!

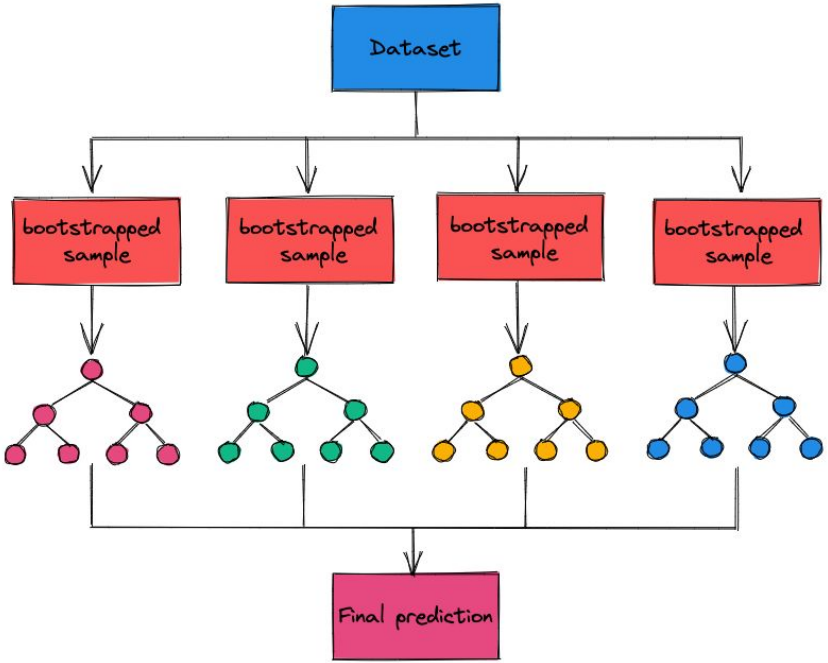
Questions?

Let’s take a 10 min break!

Random Forests

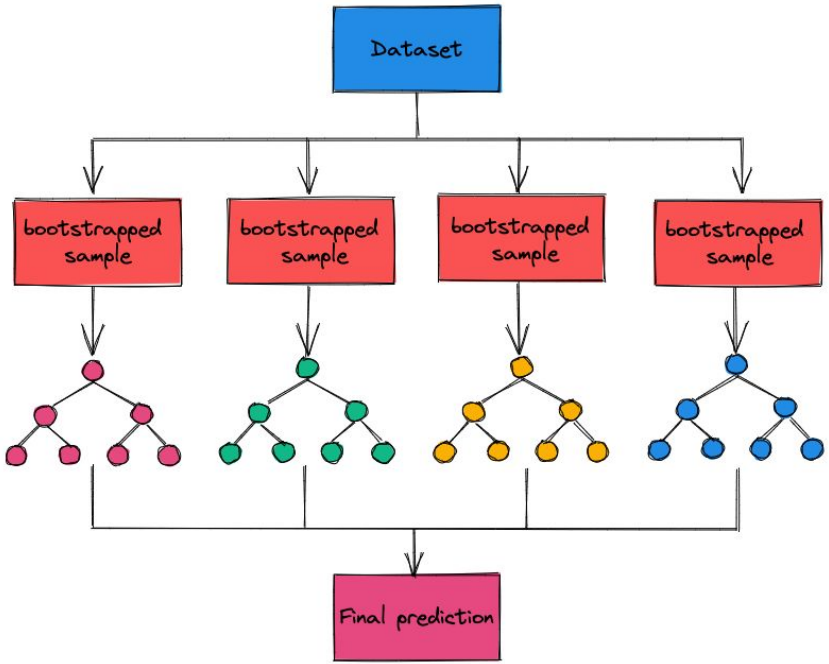
Random Forests

- Applicable to both classification and regression problems
- Smarter bagging for trees
- Motivated by theory that generalization improves with uncorrelated trees
- Bootstrapped samples and random subset of features are used to train each tree
- The outputs from each of the models are averaged to make the final prediction.



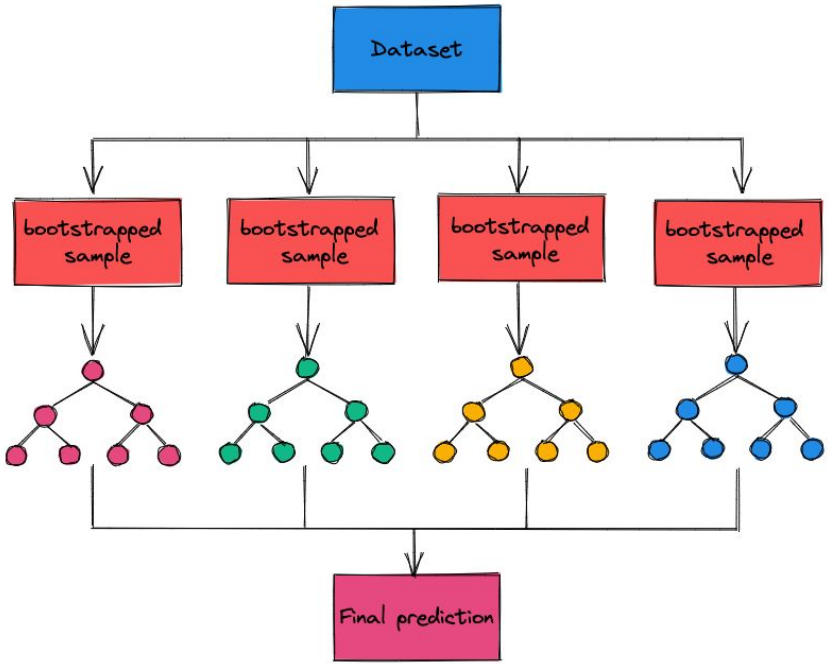
Random Forests - Hyperparameter tuning

- Random Forest hyperparameters:
  - # of trees
  - # of features
    - Classification - sqrt(# of features)
    - Regression - # of features
- Decision Tree hyperparameters (splitting criteria, maximum depth, etc.)
- Uses **out-of-bag (OOB) error** for model selection
- OOB error is the average error of a data point calculated using predictions from the trees that do not contain it in their respective bootstrap sample



Random Forests - Feature importances

- Feature importance is calculated as **the decrease in node impurity weighted by the probability of reaching that node**.
- The node probability can be calculated by the number of samples that reach the node, divided by the total number of samples.
- The higher the value the more important the feature.



Random Forests - Example

```
ohe_features = [
    "LotShape", "Street", "ExterCond", "OverallCond",
    "OverallQual", "Condition1", "Functional", "MSZoning",
    "FireplaceQu",
]
te_features = ["Neighborhood"]
num_features = [
    "BedroomAbvGr", "BsmtFinSF1", "BsmtFullBath", "EnclosedPorch",
    "GarageArea", "LotArea", "1stFlrSF", "2ndFlrSF",
    "TotalBsmtSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                     remainder="passthrough"
                       )
pipe_rf = make_pipeline(preprocess, RandomForestRegressor(random_state=100))
pipe_rf.fit(dev_X, dev_y)
print(f"Random Forest Regressor score:", pipe_rf.score(test_X, test_y))
```

```
Random Forest Regressor score: 0.8675314864296553
```

Random Forests - Example

```
ohe_features = [
    "LotShape", "Street", "ExterCond", "OverallCond",
    "OverallQual", "Condition1", "Functional", "MSZoning",
    "FireplaceQu",
]
te_features = ["Neighborhood"]
num_features = [
    "BedroomAbvGr", "BsmtFinSF1", "BsmtFullBath", "EnclosedPorch",
    "GarageArea", "LotArea", "1stFlrSF", "2ndFlrSF",
    "TotalBsmtSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                       )
pipe_rf = make_pipeline(preprocess, RandomForestRegressor(random_state=100))
pipe_rf.fit(dev_X, dev_y)
pipe_tree = make_pipeline(preprocess, DecisionTreeRegressor())
pipe_tree.fit(dev_X, dev_y)
pipe_bagging_tree = make_pipeline(preprocess, BaggingRegressor(DecisionTreeRegressor(),
                                                               n_estimators=100,
                                                               random_state=42))
pipe_bagging_tree.fit(dev_X, dev_y)
print(f"Random Forest Regressor score:", pipe_rf.score(test_X, test_y))
print(f"Decision Regressor score:", pipe_tree.score(test_X, test_y))
print(f"Bagging Tree Regressor score:", pipe_bagging_tree.score(test_X, test_y))
```

```
Random Forest Regressor score: 0.8675314864296553
Decision Regressor score: 0.7298818720921555
Bagging Tree Regressor score: 0.8616804244664917
```

Random Forests - Hyperparameter tuning

```
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                       )
n_estimators = [100, 200, 300, 400, 500]
oob_scores = []
train_scores = []
test_scores = []
for estimators in n_estimators:
    pipe_rf = make_pipeline(preprocess,
                            RandomForestRegressor(n_estimators=estimators,
                                                  random_state=100, oob_score=True))
    pipe_rf.fit(dev_X, dev_y)
    train_scores.append(pipe_rf.score(dev_X, dev_y))
    test_scores.append(pipe_rf.score(test_X, test_y))
    rf = pipe_rf.named_steps['randomforestregressor']
    oob_scores.append(rf.oob_score_)

best_index = oob_scores.index(max(oob_scores))
best_n_estimators = n_estimators[best_index]
best_pipe_rf = make_pipeline(preprocess,
                            RandomForestRegressor(n_estimators=best_n_estimators, random_state=100, oob_score=True))
best_pipe_rf.fit(dev_X, dev_y)
print(f"Random Forest Regressor score, # of estimators: {best_pipe_rf.score(test_X, test_y)}, {best_n_estimators}")
```

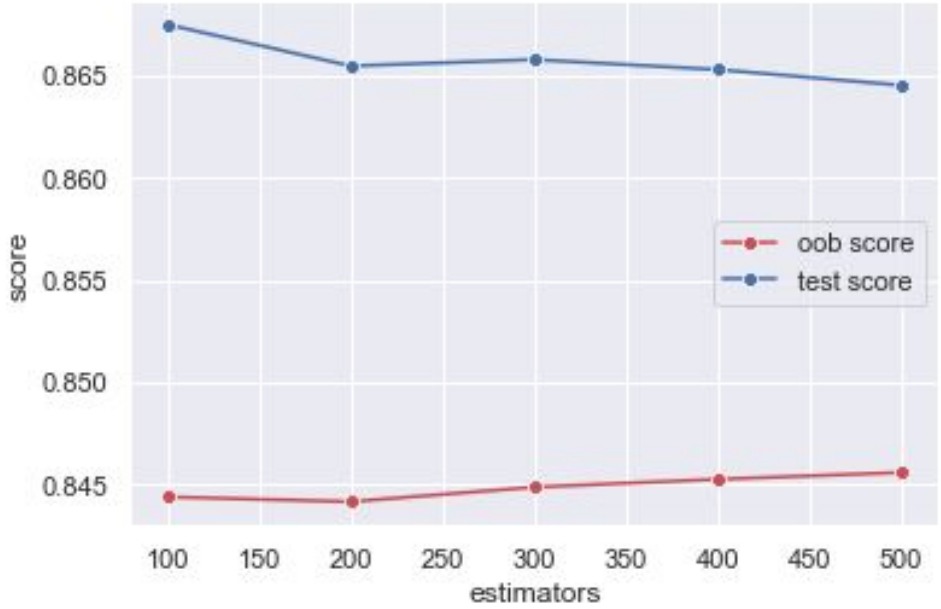
```
Random Forest Regressor score, # of estimators: 0.8645483569790224, 500
```

Random Forests - Hyperparameter tuning

```
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                       )
n_estimators = [100, 200, 300, 400, 500]
oob_scores = []
train_scores = []
test_scores = []
for estimators in n_estimators:
    pipe_rf = make_pipeline(preprocess,
                            RandomForestRegressor(n_estimators=estimators,
                                                  random_state=100, oob_score=True,
                                                  warm_start=True))
    pipe_rf.fit(dev_X, dev_y)
    train_scores.append(pipe_rf.score(dev_X, dev_y))
    test_scores.append(pipe_rf.score(test_X, test_y))
    rf = pipe_rf.named_steps['randomforestregressor']
    oob_scores.append(rf.oob_score_)

best_index = oob_scores.index(max(oob_scores))
best_n_estimators = n_estimators[best_index]
best_pipe_rf = make_pipeline(preprocess,
                            RandomForestRegressor(n_estimators=best_n_estimators, random_state=100, oob_score=True))
best_pipe_rf.fit(dev_X, dev_y)
print(f"Random Forest Regressor score, # of estimators: {best_pipe_rf.score(test_X, test_y)}, {best_n_estimators}")
```

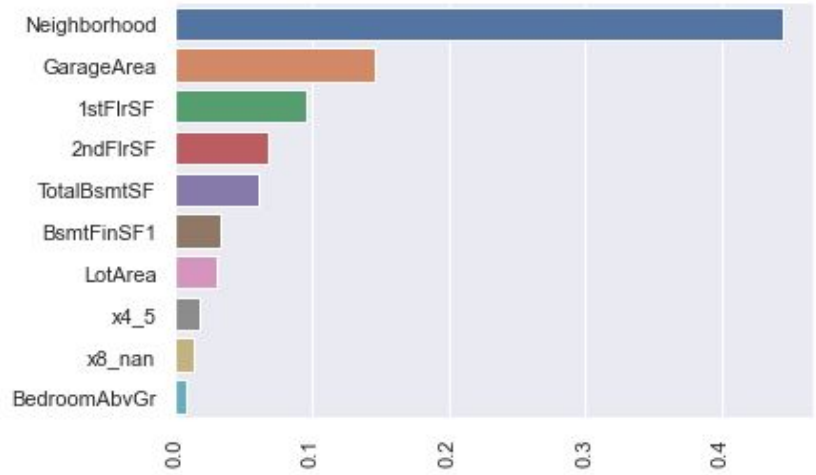
```
Random Forest Regressor score, # of estimators: 0.8645483569790224, 500
```

Random Forests - Hyperparameter tuning



Random Forests - Feature importances

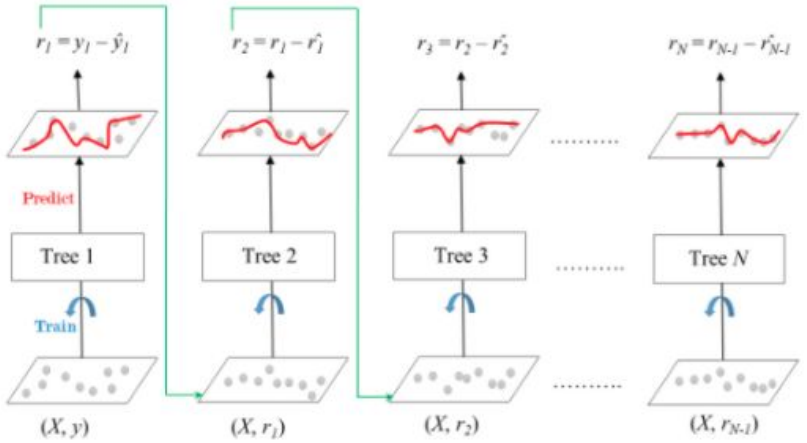
```
ohe_feature_names = preprocess.named_transformers_["onehotencoder"].get_feature_names().tolist()
te_feature_names = preprocess.named_transformers_["targetencoder"].get_feature_names()
feature_names = num_features + ohe_feature_names + te_feature_names
rf = best_pipe_rf.named_steps['randomforestregressor']
feat_imps = list(zip(feature_names, rf.feature_importances_))
feats, imps = zip(*(sorted(list(filter(lambda x: x[1] != 0, feat_imps)), key=lambda x: x[1], reverse=True)))
ax = sns.barplot(list(imps[:10]), list(feats[:10]))
ax.tick_params(axis='x', rotation=90)
```



Boosting

Boosting

- Applicable to both regression as well as classification problems
- Includes a family of ML algorithms that convert weak learners to strong ones.
- The weak learners are learned sequentially with early learners fitting simple models to the data and then analysing data for errors.
- When an input is misclassified by one tree, its output is ***adjusted*** so that next tree is more likely to learn it correctly.

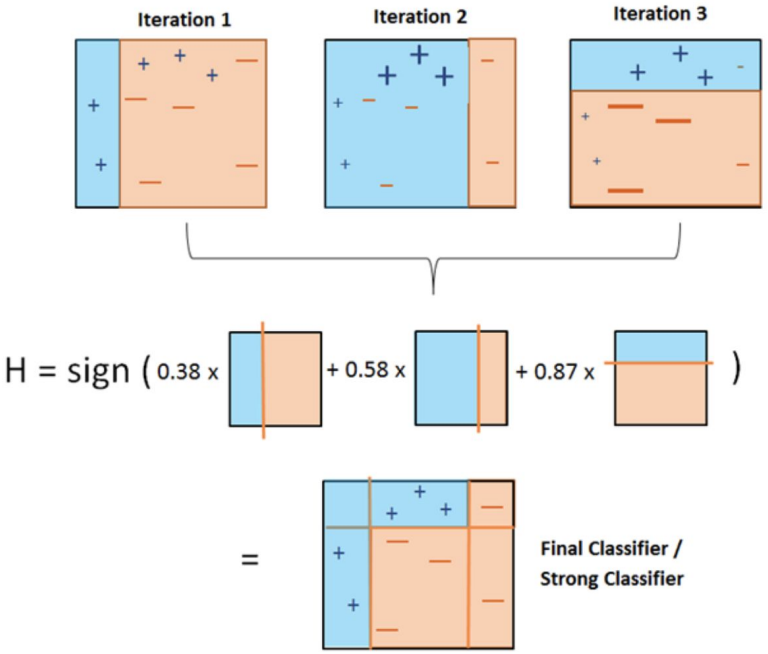


https://www.geeksforgeeks.org/ml-gradient-boosting/

Adaptive Boosting

Adaboost (Adaptive Boosting)

- First ensemble boosting algorithm applied to classification tasks
- Initially, a decision stump classifier (just splits the data into two regions) is fit to the data
- The data points correctly classified are given less weightage while misclassified data points are given higher weightage in the next iteration
- A decision stump classifier is now fit to the data with weights determined in previous iteration
- Weights (ϱ_t) for each classifier (estimated during the training process) are used to combine the outputs and make the final prediction.

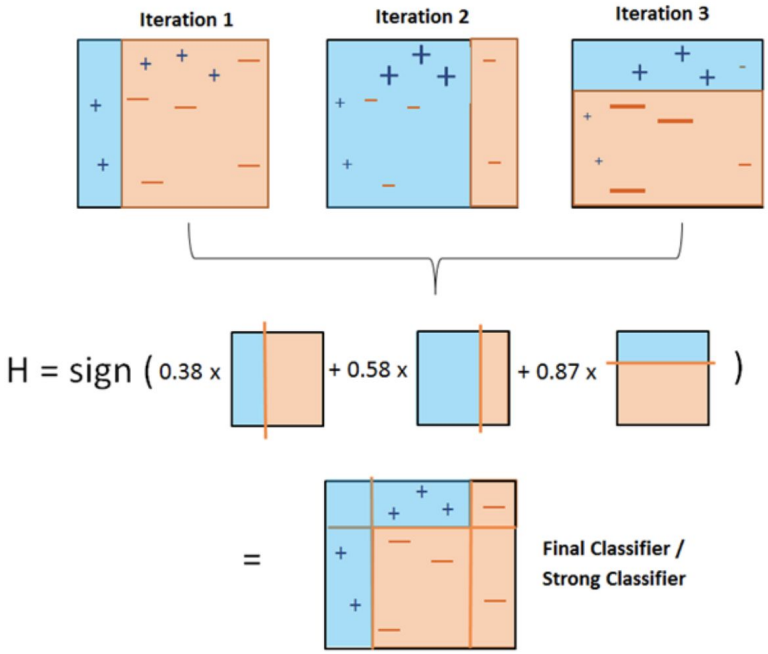


$$H(x) = sign\left[\sum_t \rho_t h_t(x)\right]$$

https://bit.ly/3kwzOxE

Adaboost Algorithm

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.



$$H(x) = sign\left[\sum_t \rho_t h_t(x)\right]$$

https://bit.ly/3kwzOxE

Understanding Adaboost

Let's first look back on how we have trained our models so far

Linear regression

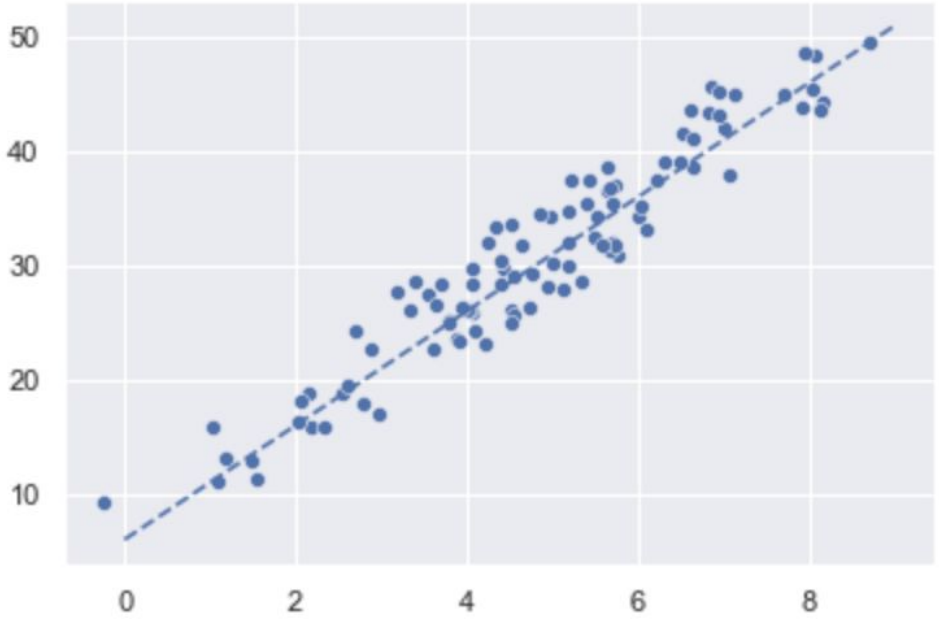
Least squares minimization

$$\underset{w}{Min} \sum_{i=1}^{m} \left(\hat{y}_i - y_i \right)^2$$

Squared-loss

$$w = (X^T X)^{-1} X^T y$$

Closed-form solution



$$\hat{y} = X\vec{w} + \vec{b}$$

$$X \in \mathbb{R}^{m \times n}, \; w \in \mathbb{R}^n$$

Logistic Regression

$$\log\left(\frac{p(y=1|x)}{p(y=-1|x)}\right) = w^T x + b$$

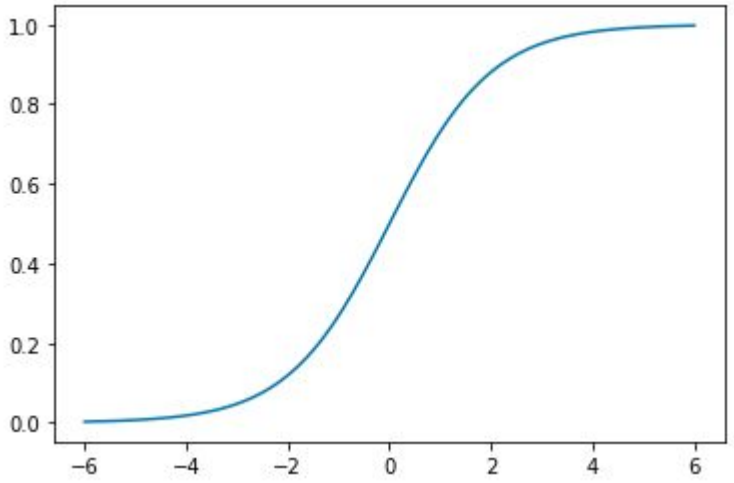
Log odds ratio

$$p(y=1|x) = \frac{1}{1 + exp(-(w^T x + b))}$$

$$Log\ Likelihood\ (LL) = \sum_{i=1}^{m} y_i \log(p_i) + (1-y_i)\log(1-p_i)$$

$$\underset{w,\ b}{Min}\ (-LL)$$

Log-loss



Understanding Adaboost

$$G(x) = \sum_{m} \alpha_m G_m(x)$$

$$L_{exp}(x, y) = exp(-yG(x))$$

$$E = \underset{\alpha_m, G_m}{Min} \left(\sum_{i} exp\left(-y_i \sum_{m} \alpha_m G_m(x_i) \right) \right)$$

Greedy optimization of exponential loss function

Understanding Adaboost

$$\frac{\partial E}{\partial \alpha_m} = 0$$

$$\alpha_m = \ln\left(\frac{1 - err_m}{err_m}\right)$$

$$err_m = \frac{\sum_{i=1}^{N} w_i I\left(y_i \neq G_m\left(x_i\right)\right)}{\sum_{i=1}^{N} w_i}$$

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Adaboost - Tuning Parameters

- Classification:
  - # estimators
  - **learning rate**
  - **base estimator**
- Regression:
  - loss function
  - learning rate
  - # of estimators
  - base estimator

Adaboost Classifier example

class `sklearn.ensemble.` **`AdaBoostClassifier`**(*base_estimator=None, *, n_estimators=50, learning_rate=1.0, algorithm='SAMME.R', random_state=None*) ¶ [source]

```
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import AdaBoostClassifier
data = load_breast_cancer()
feature_names = data.feature_names
bc_df = pd.DataFrame(data.data, columns = feature_names)
target = pd.Series(data.target)
dev_X, test_X, dev_y, test_y = train_test_split(bc_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), feature_names))
pipe_abc = make_pipeline(preprocess, AdaBoostClassifier(random_state=42))
pipe_abc.fit(dev_X, dev_y)
print(f"Adaboost Classifier score:", pipe_abc.score(test_X, test_y))
```

```
Adaboost Classifier score: 0.9736842105263158
```

Adaboost Regressor example

class `sklearn.ensemble.` **`AdaBoostRegressor`**(*base_estimator=None*, ***, *n_estimators=50*, *learning_rate=1.0*, *loss='linear'*, *random_state=None*) [source]

```
from sklearn.ensemble import AdaBoostRegressor
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                         (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                         (TargetEncoder(handle_unknown="ignore"), te_features),
                                      remainder="passthrough"
                        )
pipe_ab = make_pipeline(preprocess, AdaBoostRegressor(random_state=42))
pipe_ab.fit(dev_X, dev_y)
print(f"Adaboost Regressor score:", pipe_ab.score(test_X, test_y))
```

```
Adaboost Regressor score: 0.7713074550934114
```

Adaboost Regressor example

```
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                       )
pipe_abr = make_pipeline(preprocess, AdaBoostRegressor(random_state=42))
pipe_abr.fit(dev_X, dev_y)
pipe_tree = make_pipeline(preprocess, DecisionTreeRegressor())
pipe_tree.fit(dev_X, dev_y)
pipe_bagging_tree = make_pipeline(preprocess, BaggingRegressor(DecisionTreeRegressor(),
                                                               n_estimators=50,
                                                               random_state=42))
pipe_bagging_tree.fit(dev_X, dev_y)
print(f"Adaboost Regressor score:", pipe_abr.score(test_X, test_y))
print(f"Decision Regressor score:", pipe_tree.score(test_X, test_y))
print(f"Bagging Tree Regressor score:", pipe_bagging_tree.score(test_X, test_y))
```

```
Adaboost Regressor score: 0.7713074550934114
Decision Regressor score: 0.7364342104167236
Bagging Tree Regressor score: 0.860669549121929
```

Adaboost Regressor example

```
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                       )
pipe_abr = make_pipeline(preprocess, AdaBoostRegressor(DecisionTreeRegressor(), random_state=42))
pipe_abr.fit(dev_X, dev_y)
pipe_tree = make_pipeline(preprocess, DecisionTreeRegressor())
pipe_tree.fit(dev_X, dev_y)
pipe_bagging_tree = make_pipeline(preprocess, BaggingRegressor(DecisionTreeRegressor(),
                                                               n_estimators=50,
                                                               random_state=42))
pipe_bagging_tree.fit(dev_X, dev_y)
print(f"Adaboost Regressor score:", pipe_abr.score(test_X, test_y))
print(f"Decision Regressor score:", pipe_tree.score(test_X, test_y))
print(f"Bagging Tree Regressor score:", pipe_bagging_tree.score(test_X, test_y))
```

```
Adaboost Regressor score: 0.8657990175363661
Decision Regressor score: 0.7582529749565105
Bagging Tree Regressor score: 0.860669549121929
```

Adaboost Regressor - hyperparameter tuning

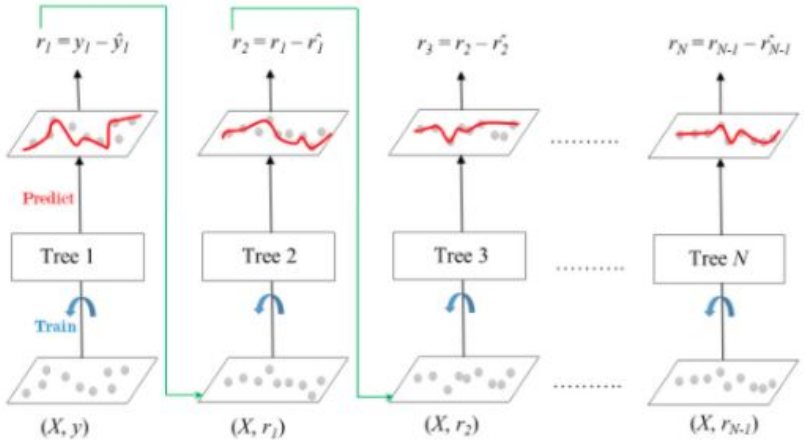
```
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                         (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                         (TargetEncoder(handle_unknown="ignore"), te_features),
                                     remainder="passthrough"
                        )
pipe = make_pipeline(preprocess,
                     GridSearchCV(AdaBoostRegressor(DecisionTreeRegressor(), random_state=42),
                                  param_grid = [{"n_estimators":[50, 100, 200],
                                                 "learning_rate":np.logspace(-3, 0, 4)}]))
pipe.fit(dev_X, dev_y)
grid_search_results = pipe.named_steps["gridsearchcv"]
print(f"Best score:", grid_search_results.best_score_)
print(f"Best alpha:", grid_search_results.best_params_)
print(f"Test score:", pipe.score(test_X, test_y))
```

```
Best score: 0.8458024295205678
Best alpha: {'learning_rate': 1.0, 'n_estimators': 200}
Test score: 0.8669448660270312
```

Gradient Boosting

Gradient Boosting

- Works for both classification and regression tasks
- Trains regression trees in a sequential manner on modified versions of the datasets.
- Every tree is trained on the residuals of the data points obtained by subtracting the predictions from the previous tree.
- Weights for each classifier (estimated during the training process) are used to combine the outputs and make the final prediction.



Gradient Boosting Algorithm

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

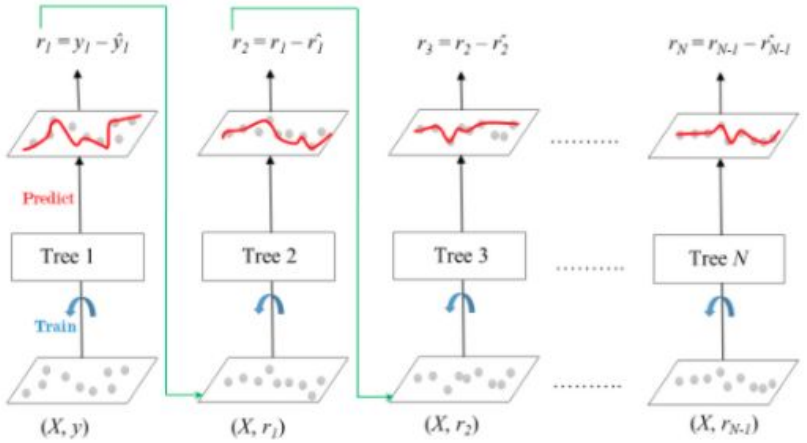
    2. Fit a base learner (or weak learner, e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.



https://www.geeksforgeeks.org/ml-gradient-boosting/

Understanding Gradient Boosting

Understanding Gradient Boosting

We need to take a detour first

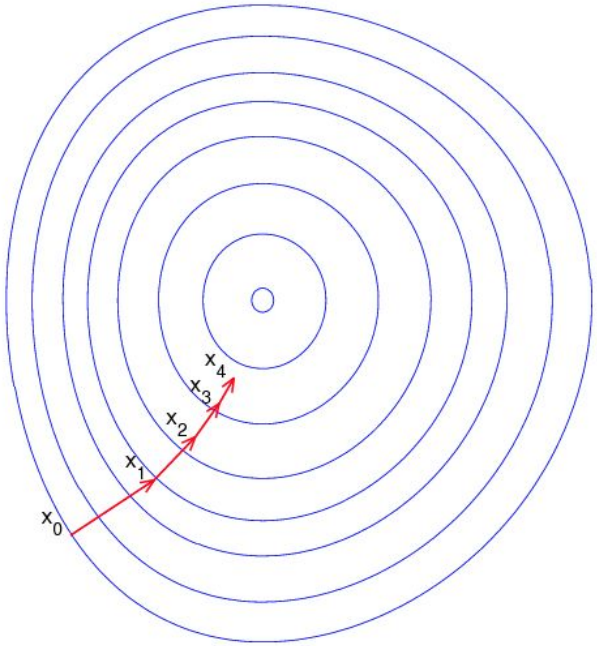
Gradient Descent

$$\underset{w}{Min}\, f(w)$$

$$w^{(i+1)} = w^{i} - \eta \nabla f(w^{i})$$

learning rate

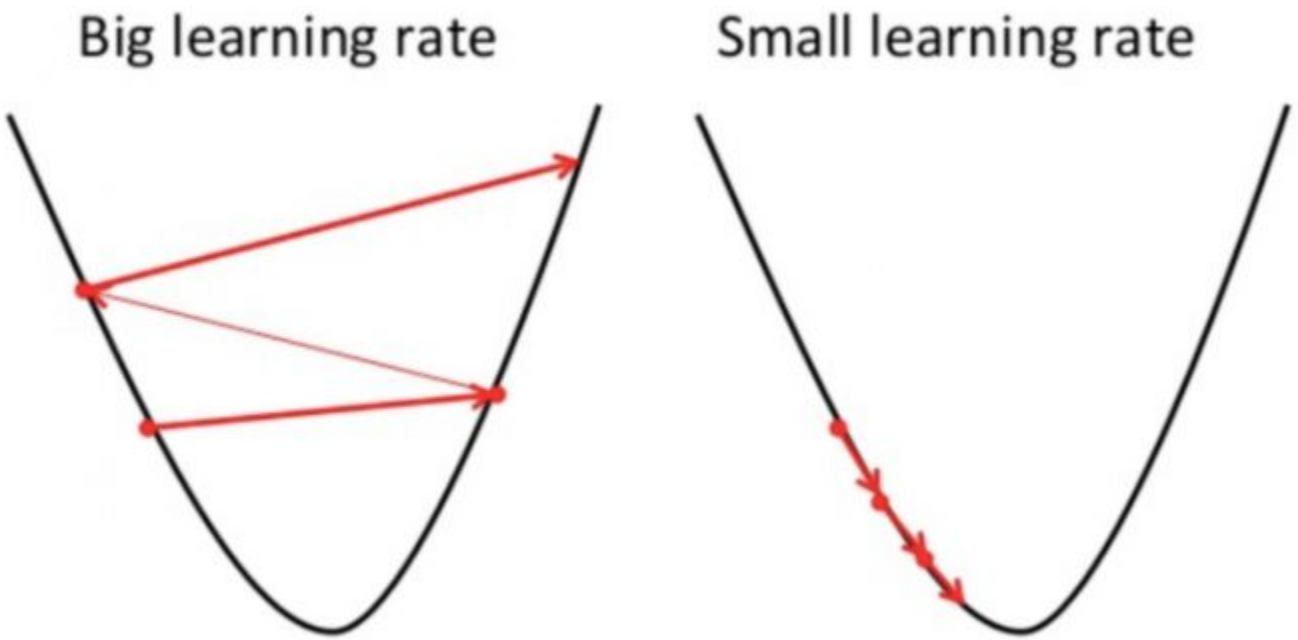
Move in the direction of steepest descent



converges to a local minimum

https://builtin.com/data-science/gradient-descent

Gradient Descent - learning rate



https://builtin.com/data-science/gradient-descent

Linear regression

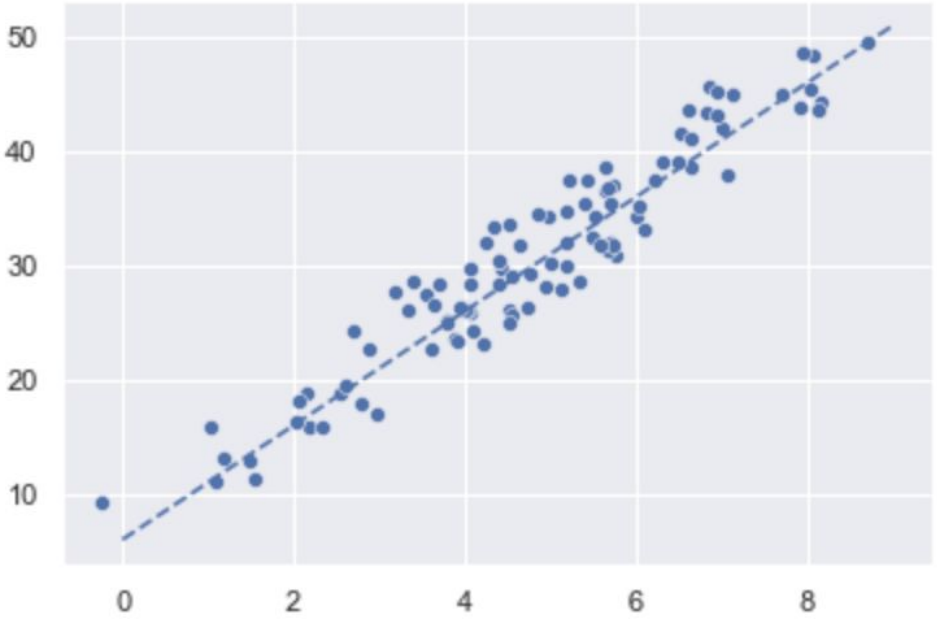
Least squares minimization

$$\underset{w}{Min} \sum_{i=1}^{m} \left(\hat{y}_i - y_i \right)^2$$

Squared-loss

$$L(w, b) = \underset{w}{Min} \sum_{i=1}^{m} \left(w^T x_i + b - y_i \right)^2$$

$$\frac{\partial L}{\partial w} = 0; \quad \frac{\partial L}{\partial b} = 0$$



$$\hat{y} = X\vec{w} + \vec{b}$$

$$X \in \mathbb{R}^{m \times n}, \; w \in \mathbb{R}^n$$

Understanding Gradient Boosting

$$F(x) = \sum_{m} \gamma_m F_m(x)$$

$$L(x, y) = \frac{1}{2}(y - F(x))^2$$

$$E = \min_{\gamma_m, F_m} \left(\frac{1}{2} \sum_{i} \left(y_i - F(x_i) \right)^2 \right)$$

Understanding Gradient Boosting

$$F_m(x) = F_{m-1}(x) - \gamma \frac{\partial E}{\partial F_{m-1}(x)}$$

$$w^{(i+1)} = w^i - \eta \nabla f(w^i)$$

$$\frac{\partial E}{\partial F_{m-1}(x)} = y - F_{m-1}(x)$$

residual

Input: training set $\{(x_i, y_i)\}_{i=1}^n$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_\gamma \sum_{i=1}^n L(y_i, \gamma).$$

2. For m = 1 to M:
   1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^n$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_\gamma \sum_{i=1}^n L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Gradient Boosting - Tuning Parameters

- # of estimators
- Learning rate
- Decision tree parameters (max depth, min number of samples etc.)
- Regularization parameters
- Row sampling
- Column sampling

Gradient Boosting implementations

- GradientBoostingClassifier
- HistGradientBoostingClassifier
- XGBoost
- CatBoost
- LightGBM

GradientBoostingClassifier

- Early implementation of Gradient Boosting in sklearn
- Typical slow on large datasets
- Most important parameters are # of estimators and learning rate
- Supports both binary & multi-class classification
- Supports sparse data

HistGradientBoostingClassifier

- Orders of magnitude faster than GradientBoostingClassifier on large datasets
- Inspired by LightGBM implementation
- Histogram-based split finding in tree learning
- Does not support sparse data
- Supports both binary & multi-class classification
- Natively supports categorical features
- Does not support monotonicity constraints

XGBoost

- One of most popular implementations of gradient boosting
- Fast approximate split finding based on histograms
- Supports GPU training, sparse data & missing values
- Adds l1 and l2 penalties on leaf weights
- Monotonicity & feature interaction constraints
- Works well with pipelines in sklearn due to a compatible interface
- Does not support categorical variables natively

https://xgboost.readthedocs.io/en/latest/python/python_intro.html

LightGBM

- Supports GPU training, sparse data & missing values
- Histogram-based node splitting
- Use Gradient-based One-Sided Sampling (GOSS) for tree learning
- Exclusive feature bundling to handle sparse features
- Generally faster than XGBoost on CPUs
- Supports distributed training on different frameworks like Ray, Spark, Dask etc.
- CLI version

https://lightgbm.readthedocs.io/en/latest/Features.html

CatBoost

- Optimized for categorical features
- Uses target encoding to handle categorical features
- Uses ordered boosting to build “symmetric” trees
- Overfitting detector
- Tooling support (Jupyter notebook & Tensorboard visualization)
- Supports GPU training, sparse data & missing values
- Monotonicity constraints

https://catboost.ai/docs/concepts/about.html

GradientBoostingRegressor - Example

```
from sklearn.ensemble import GradientBoostingRegressor
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                         (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                         (TargetEncoder(handle_unknown="ignore"), te_features),
                                     remainder="passthrough"
                        )
pipe_gbr = make_pipeline(preprocess, GradientBoostingRegressor(random_state=42))
pipe_gbr.fit(dev_X, dev_y)
print(f"Gradient Boosting Regressor score:", pipe_gbr.score(test_X, test_y))
```

```
Gradient Boosting Regressor score: 0.8874253076454832
```

HistGradientBoostingClassifier - Example

```
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.base import TransformerMixin

class DenseTransformer(TransformerMixin):
    def fit(self, X, y=None, **fit_params):
        return self

    def transform(self, X, y=None, **fit_params):
        return X.todense()

mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                       )
pipe_hgbr = make_pipeline(preprocess, DenseTransformer(), HistGradientBoostingRegressor(random_state=42))
pipe_hgbr.fit(dev_X, dev_y)
print(f"Histogram Gradient Boosting Regressor score:", pipe_hgbr.score(test_X, test_y))
```

```
Histogram Gradient Boosting Regressor score: 0.8801313680707443
```

XGBoost - Example

```
from xgboost import XGBRegressor
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                         (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                         (TargetEncoder(handle_unknown="ignore"), te_features),
                                     remainder="passthrough"
                        )
pipe_xgb = make_pipeline(preprocess, XGBRegressor(random_state=42))
pipe_xgb.fit(dev_X, dev_y)
print(f"XGBoost Regressor score:", pipe_xgb.score(test_X, test_y))
```

```
XGBoost Regressor score: 0.8684139695772952
```

LightGBM - Example

```
from lightgbm import LGBMRegressor
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                         (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                         (TargetEncoder(handle_unknown="ignore"), te_features),
                                     remainder="passthrough"
                        )
pipe_xgb = make_pipeline(preprocess, LGBMRegressor(random_state=42))
pipe_xgb.fit(dev_X, dev_y)
print(f"LightGBM Regressor score:", pipe_xgb.score(test_X, test_y))
```

```
LightGBM Regressor score: 0.8829269193577356
```

CatBoost - Example

```
from catboost import CatBoostRegressor
mixed_df = data_df[ohe_features + te_features + num_features]
target = np.log(data_df["SalePrice"])
dev_X, test_X, dev_y, test_y = train_test_split(mixed_df, target,
                                                test_size=0.2, random_state=42)
preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                       )
pipe_cbr = make_pipeline(preprocess, CatBoostRegressor(random_state=42, verbose=False))
pipe_cbr.fit(dev_X, dev_y)
print(f"CatBoost Regressor score:", pipe_cbr.score(test_X, test_y))
```

```
CatBoost Regressor score: 0.9128081167510569
```

Questions?