CS37300 Data Mining & Machine Learning

Bruno Ribeiro

Complex Models for Large Datasets

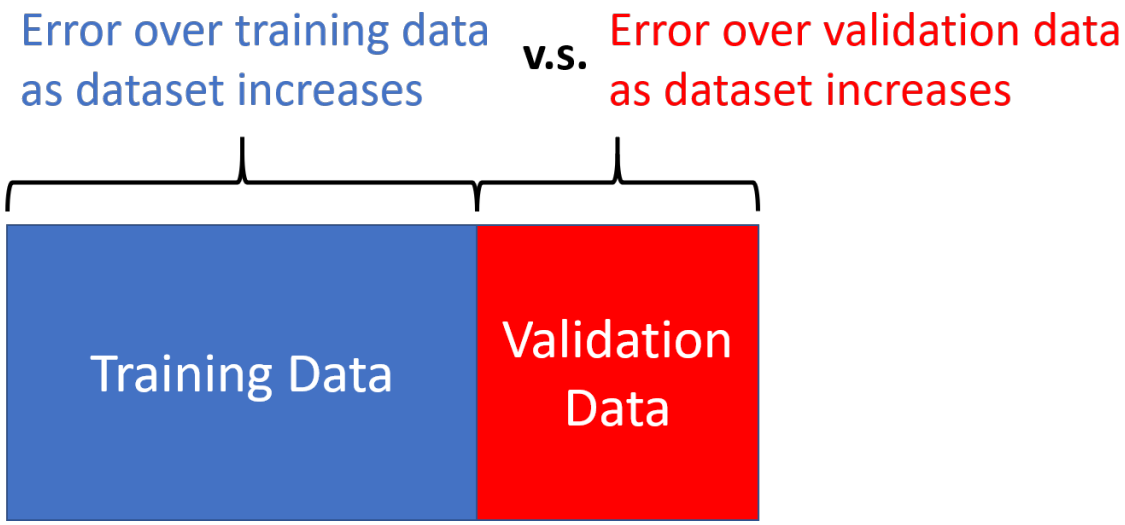
Boosted Decision Trees

Learning Curves

A learning curve is a graphical representation of how an increase in amount of data improves the model:

- **error** (measured on the **vertical axis**) as a function of
- **training examples** (the **horizontal axis**)

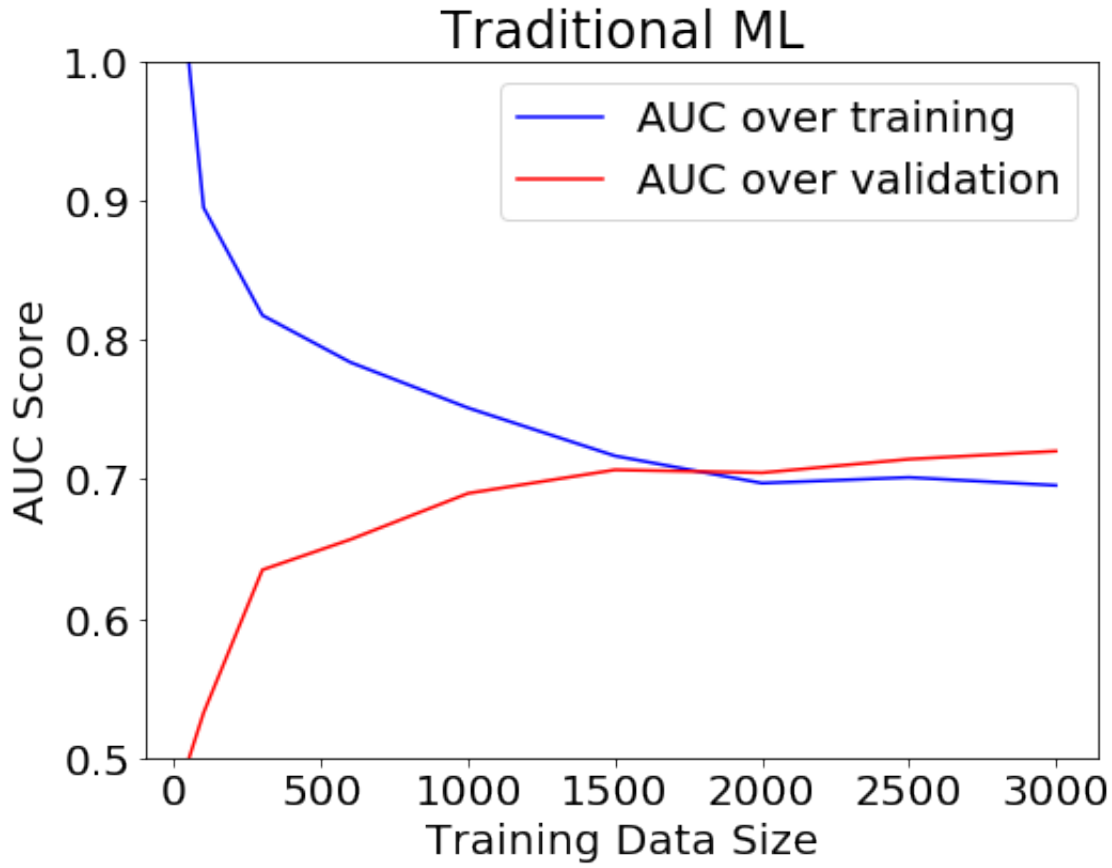
- It is measured both w.r.t. to the **training dataset** v.s. a separate **validation dataset**.



- Important:
  - **The gap between the error over the training data and error over validation data is a good proxy to assess the bias and variance of a model**

Learning curves and generalization error

- The lerning curve of a Logistic Regression model (supervised classification task):



- If the gap between validation and training closes, the model error is likely only noise + bias

- How fast the gap closes, indicates the amount of bias
  - Faster closing = large bias
  - Slower closing = small bias

More Data Requires More Complex Models

In today's class we will study boosted decision trees.

- Boosted decision trees are built using compositions
  - Boosted decision trees can be as complex or as simple as we want

Gradient boosted decision trees are **the dominant method for classification and regression of structured data**

- Structured data is any data whose feature vectors are obtained directly from the data

- Examples include input data such as $\mathbf{x} =$(age, gender, annual income, FICO scores)
  - The features must be discrete or easily discretized (e.g., create binary features based on quantiles)
    - For instance, age can be discretized by multiple binary variables describing age ranges: $Q_{18\ldots21} \in \{0,1\}, Q_{22\ldots28} \in \{0,1\}, \ldots$.
  - These features are widely used in applications bank loan repayment predictions, insurance premiums values, airline ticket prices, etc..

- Images, free text, and graphs are considered **unstructured data**, since obtaining features from these inputs is, in itself, a learning task
  - Neural networks much better at unstructured data tasks, since a Gradient boosted decision tree is as good as the method that creates the features

Supervised Learning Review

Notation

$(x_i, y_i) \in \mathbb{R}$ is the i-th training example tuple, where x_i is its set of features and y_i its label.

The training data is denoted by $D_{\text{train}} = \{(x_i, y_i)\}_{i=1}^{N}$.

Score Function (Model Evaluation)

Let

$$\hat{y}_i = f(x_i; \mathbf{W})$$

be our model prediction of the label of example i, where $\mathbf{W}$ are the parameters of our model.

The score function is our way to evaluate the goodness of a model:

$$F(D_{\text{train}}; \mathbf{W}) = \underbrace{L(D_{\text{train}}; \mathbf{W})}_{\substack{\text{Training score} \\ \text{measures how well} \\ \text{model describes } D_{\text{train}}}} + \underbrace{\Omega(\mathbf{W})}_{\substack{\text{Penalty,} \\ \text{for model complexity}}},$$

where

$$L(D_{\text{train}}; \mathbf{W}) = \sum_{i=1}^{N} l(y_i, \hat{y}_i).$$

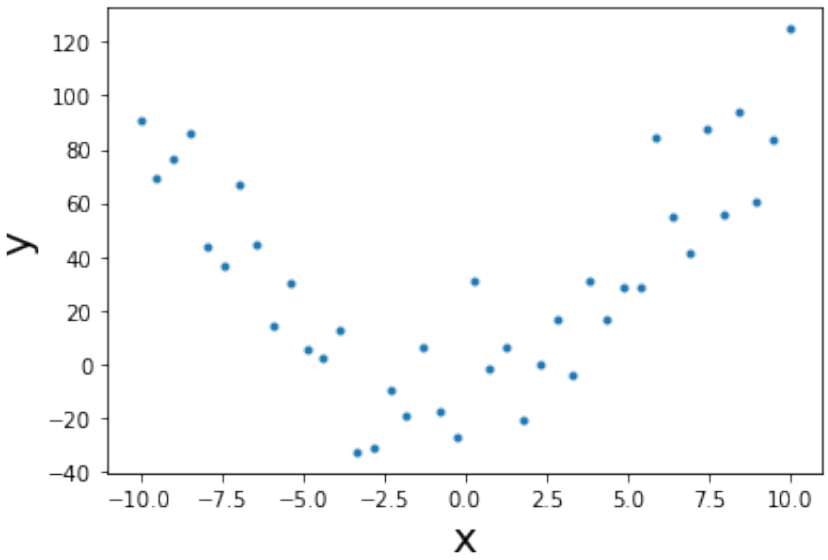
Examples of score functions and model penalties

- Training score:
  - Square loss: $l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$
  - Negative log-likelihood (Cross-entropy loss for binary labels): $l(y_i, \hat{y}_i) = -y_i \log(\hat{y}_i) - (1 - y_i) \log(1 - \hat{y}_i)$
- Penalty for model complexity, for some constant $\lambda > 0$ (we covered these in a previous class):
  - L2 norm: $\Omega(\mathbf{w}) = \lambda \|\mathbf{w}\|^2$
  - L1 norm: $\Omega(\mathbf{w}) = \lambda \|\mathbf{w}\|_1$

Consider 1D Regression Problem (x,y)

In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

# No randomness
np.random.seed(42)

#Generate N points for training data
N = 40

# get 1000 equally spaced points between -10 and 10
x = np.linspace(-10, 10, N)

# calculate the y value for each element of the x vector
y = x**2 + 2*x + 1 + np.random.normal(0,20,N)

def plotdata():
    fig, ax = plt.subplots()
    plt.xlabel("x", fontsize=20)
    plt.ylabel("y", fontsize=20)
    ax.plot(x, y,'.')

plotdata()
plt.show()
```

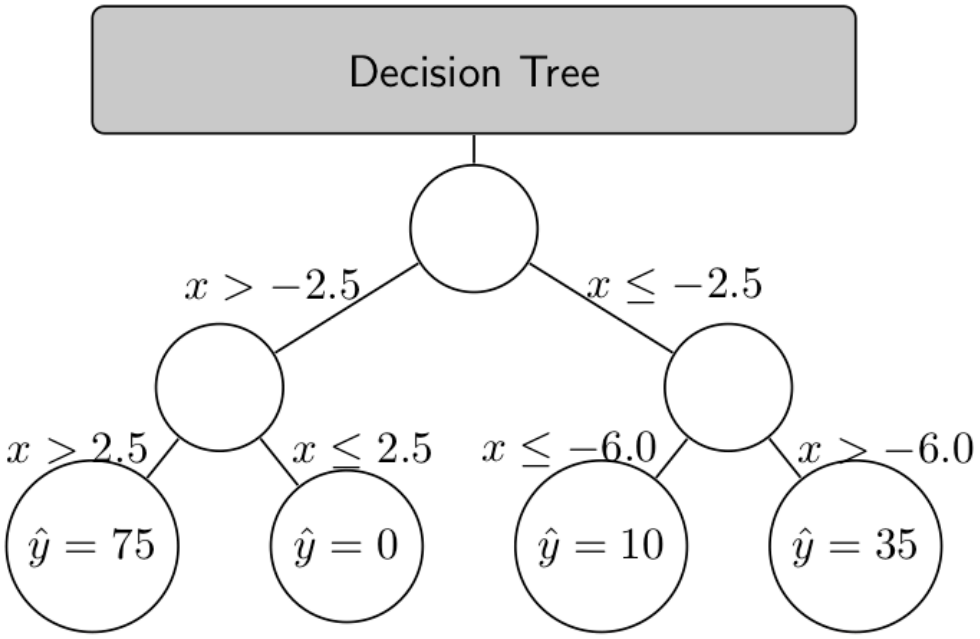


Regression Tree (CART) Review

Regression tree (also known as classification and regression tree) is a set of decision rules (a decision tree) that contains a regression prediction at each leaf node.

- Often, we will build the tree to just have a single constant prediction at the leaf node

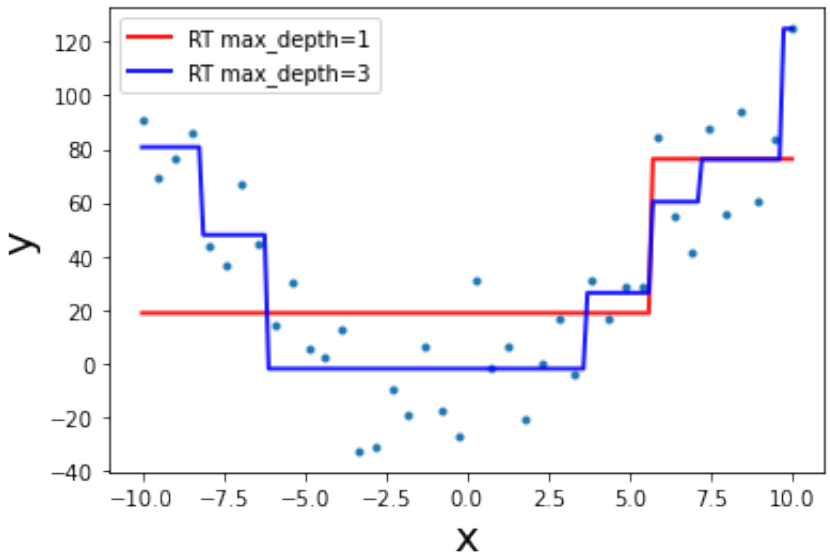
Example of tree for regression problem given above



CART Example

In [2]:

```
# Example of a simple regression tree (no boosting or gradient boosting)
from sklearn.tree import DecisionTreeRegressor

#Reshape x:
x_train = x[:, np.newaxis]
y_train = y[:, np.newaxis]

new_x = np.linspace(-10, 10, 4*N)

plotdata()
DT = DecisionTreeRegressor(max_depth=1).fit(x_train, y_train)
plt.plot(new_x, DT.predict(new_x[:, np.newaxis]),
         label='RT max_depth=1', color='r', alpha=0.9, linewidth=2)

DT = DecisionTreeRegressor(max_depth=3).fit(x_train, y_train)
plt.plot(new_x, DT.predict(new_x[:, np.newaxis]),
         label='RT max_depth=3', color='b', alpha=0.9, linewidth=2)

plt.legend(loc='upper left')
plt.show()
```



Tree Ensembles

Thinking of a regression tree as a function f that maps the attributes to a value:

Model: assuming we have K trees, $f_1, \ldots, f_K$,

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in \mathcal{F},$$

where $\mathcal{F}$ is the space **of all** functions containing **all regression trees**.

Thinking of tree search as an optimization problem:

$$\mathbf{F}^{\star} = \arg\min_{\mathbf{F}} F(D_{\text{train}}; \mathbf{F}) = \arg\min_{\mathbf{F}} \sum_{i=1}^{N} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

where

$$\mathbf{F} = (f_1, \ldots, f_K)$$

and $\Omega(f_k)$ is a penalty for **tree complexity**.

Possible ways to define penalty Ω?

- Number of nodes in the tree
- Tree depth
- Square value of the leaf weights

Challenge with the above optimization?

Learning the simplest (smallest) decision tree is an NP-complete problem [Hyafil & Rivest '76]

Trees: Optimization vs Heuristics

- When we talk about learning decision trees, we are often refering to heuristics
  - Split a node by some information criteria (e.g., information gain, GINI index)

Boosting (sequential additive training)

Our **real** objective is still

$$\mathbf{F}^{\star} = \arg\min_{\mathbf{F}} \sum_{i=1}^{N} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

but we still cannot use gradient descent to find $\mathbf{F}^{\star}$ since the search space are trees, not differentiable functions...

...but there is another approximation we can make: **Additive training (Boosting)**

Boosting (Sequential Additive Training)

Start from constant prediction, add new functions sequentially. Let $\hat{y}_i^{(k)}$ be the prediction of y_i at the k-th iteration of our algorithm.

$$\hat{y}_i^{(0)} = 0 \tag{1}$$
$$\hat{y}_i^{(1)} = \hat{y}_i^{(0)} + \alpha f_1(x_i) \tag{2}$$
$$\vdots \tag{3}$$
$$\hat{y}_i^{(K)} = \hat{y}_i^{(K-1)} + \alpha f_K(x_i), \tag{4}$$

where $\alpha > 0$ is what we call a "**learning rate**".

- Note that the learning rate α shrinks the contribution of each tree by α.
  - To increase the number of trees K we should decrease the learning rate α, otherwise overfitting could be an issue.

How can we decide which f to add at each iteration?

Look at the objective function.

$$\mathbf{F}^\star = \arg\min_{\mathbf{F}} \sum_{i=1}^{N} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(\alpha f_k)$$

We need to add f_k to the prediction

$$\hat{y}_i^{(k)} = \hat{y}_i^{(k-1)} + \alpha f_k(x_i)$$

that will minimize the score.

The score function at the k-th step is:

$$\text{Score}^{(k)} = \sum_{i=1}^{N} l\left(y_i, \hat{y}_i^{(k-1)} + \alpha f_k(x_i)\right) + \sum_{m=1}^{k} \Omega(\alpha f_m)$$

Square loss example

If l is the square loss, then the score is

$$\begin{aligned}\text{Score}^{(k)} &= \sum_{i=1}^{N} l\left(y_i, \hat{y}_i^{(k-1)} + \alpha f_k(x_i)\right) + \sum_{m=1}^{k} \Omega(\alpha f_m) \\ &= \sum_{i=1}^{N} \left(y_i - \hat{y}_i^{(k-1)} - \alpha f_k(x_i)\right)^2 + \sum_{m=1}^{k} \Omega(\alpha f_m) \\ &= \sum_{i=1}^{N} \left(-2(y_i - \hat{y}_i^{(k-1)})\alpha f_k(x_i) + \alpha^2 f_k(x_i)^2\right) + \sum_{m=1}^{k} \Omega(\alpha f_m) + \text{constant that d} \\ &= \sum_{i=1}^{N} \left(h_i^{(k)} \alpha f_k(x_i) + \alpha^2 f_k(x_i)^2\right) + \sum_{m=1}^{k} \Omega(\alpha f_m) + \text{constant that does not dep}\end{aligned}$$

where $h_i^{(k)} = 2(\hat{y}_i^{(k-1)} - y_i)$ is called the residual of round $k-1$.

Translation: The problem now becomes one of finding a tree f_k that **cares** about the correct prediction of example i with a weight proportional to the residual $h_i^{(k)}$, while trying to keep the overall predicted values not too large due to the $\alpha^2 f_k(x_i)^2$ term.

This decomposition works for the square loss, but what about cross-entropy or other forms of losses?

What about other loss functions?

$$\text{Score}^{(k)} = \sum_{i=1}^{N} l\left(y_i, \hat{y}_i^{(k-1)} + \alpha f_k(x_i)\right) + \sum_{m=1}^{k} \Omega(\alpha f_m)$$

- Consider the Taylor expansion of the score:
  - Recall that $f(x+\Delta x) \approx f(x) + \frac{d}{dx} f(x) \Delta x$
  - Define

$$h_i^{(k)} = \frac{\partial}{\partial \hat{y}^{(k-1)}} l(y_i, \hat{y}^{(k-1)})$$

  - Then, the Taylor series approximation yields (math details at the end of the lecture):

$$\begin{aligned} \text{Score}^{(k)} &\approx \sum_{i=1}^{N} \left(l(y_i, \hat{y}_i^{(k-1)}) + \alpha h_i^{(k)} f_k(x_i) \right) + \sum_{m=1}^{k} \Omega(\alpha f_m) \\ &\approx \sum_{i=1}^{N} h_i^{(k)} \alpha f_k(x_i) + \Omega(\alpha f_k) + \text{constant that does not depend on } f_k \end{aligned}$$

    where in the last equation we used the fact that at the k-th iteration, both $l(y_i, \hat{y}_i^{(k-1)})$ and $\sum_{m=1}^{k-1} \Omega(\alpha f_m)$ are constant.

Decision Tree objective of Tree f_k

Decision tree f_k must minimize

$$f_k^\star = \arg\min_{f_k} \sum_{i=1}^{N} h_i^{(k)} \alpha f_k(x_i) + \alpha^2 f_k(x_i)^2 + \Omega(\alpha f_k)$$

We can use any decision tree algorithm to greedly learn f_k (it will likely not be optimal), weighted by the residual $h_i^{(k)}$.

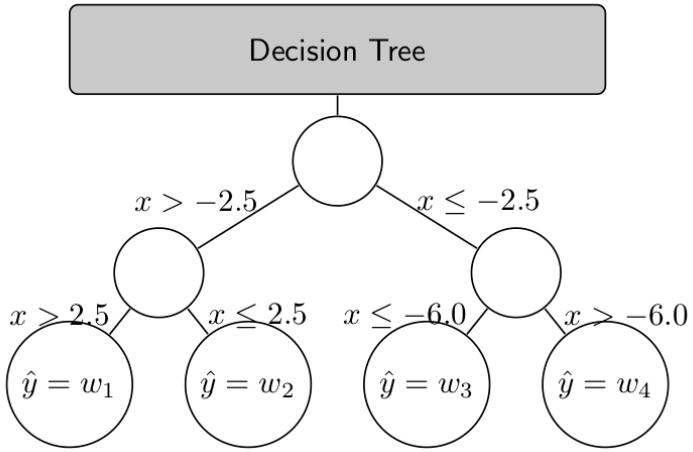
- In what follows, assume f_k is known (has been learned)

How can we further improve gradient boosting for trees?

So far, at each leaf gives a constant value, determined while we build the tree.

Insight: After we learn the tree structure $q^{(k)}(\cdot)$, we can use gradient descent to improve the leaf weights to further reduce the loss over the training data.

In what follows, we decouple these two procedures: learning the tree and giving a prediction



once we reach a leaf.

- Suppose we have learned tree f_k, but treat leaf weights $w_1, \ldots, w_J$ as paramters
- Let $q^{(k)}(x_i)$ be the leaf index of example x_i on tree f_k, once we follow the tree.
  - This gives the tree structure, but not its leaf values
- We define the output of leaf $q^{(k)}(x_i)$ of tree f_k as a parameter $w_{q^{(k)}(x_i)} \in \mathbb{R}$ that we will also learn.
- Let J be the number of leaves of f_k (denote $J^{(k)}$ if J varies for different trees f_k).
- Let $\mathbf{w}$ be a J-dimensional vector with all the leaf outputs.

We can replace $f_k(x_i) = \mathbf{w}_{q^{(k)}(x_i)}$ in the vector $\mathbf{w}$.

Hence, we can rewrite our score function

- Let $I_j^{(k)} = \{i : q^{(k)}(x_i) = j\}$ be the set of training examples whose output $f_k(x_i)$ comes from leaf node $j \in \{1, \ldots, J\}$.
- Then

$$\begin{aligned}\text{Score}^{(k)} &\approx \sum_{i=1}^{N} h_i^{(k)} \alpha f_k(x_i) + \alpha^2 f_k(x_i)^2 + \sum_{m=1}^{k} \Omega(\alpha f_m) + \text{constant that does not de} \\ &\approx \sum_{j=1}^{J} \left(\sum_{i \in I_j} (h_i^{(k)} + \alpha \mathbf{w}_j) \right) \alpha \mathbf{w}_j + \Omega(\alpha f_k) + \text{constant that does not depen}\end{aligned}$$

And note that $\left(\sum_{i \in I_j} h_i\right)$ is only a function of w_j. Then,

$$\begin{aligned}\nabla_{\mathbf{w}} \text{Score}^{(k)} &\approx \nabla_{\mathbf{w}} \left(\sum_{j=1}^{J} \left(\sum_{i \in I_j} h_i^{(k)} + \alpha \mathbf{w}_j \right) \alpha \mathbf{w}_j + \Omega(\alpha f_k) \right) \\ &\approx \sum_{j=1}^{J} \frac{\partial}{\partial \mathbf{w}_j} \left(\sum_{i \in I_j} h_i^{(k)} + \alpha \mathbf{w}_j \right) \alpha \mathbf{w}_j + \nabla_{\mathbf{w}} \Omega(\alpha f_k)\end{aligned}$$

That is, after we learn the tree structure that gives $q^{(k)}(\cdot)$, we can use gradient descent to improve the leaf weights to further reduce the loss over the training data.

For arbitrary positive values of $\gamma > 0$ and $\lambda > 0$, one possible definition of tree complexity is

$$\Omega(f_k) = \gamma J + \frac{1}{2} \lambda \alpha \sum_{j=1}^{J} \mathbf{w}_j^2,$$

where $\mathbf{w}_j$ is the prediction given by leaf j, i.e., $\mathbf{w}_{q^{(k)}(x_i)} = f_k(x_i)$.

Properties

- There is a trade-off between the "learning rate" α and the number of boosted trees K
- Gradient boosting is relatively robust to overfitting if we increase K as long as we reduce α
  - Each tree should only have a small impact in the overall prediction.
- For structured data, e.g., Bank Loan applications, boosted decision trees are currently the best predictor

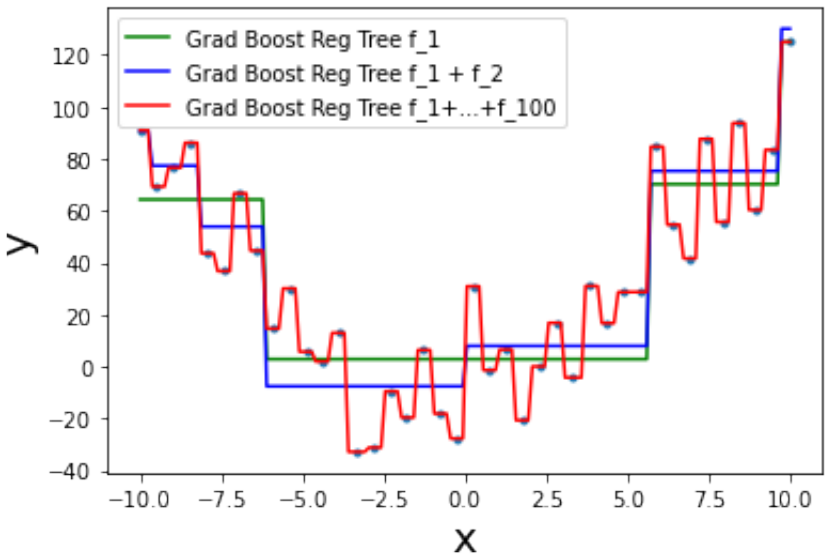
Scalability

- The sequential nature of the boosted decision tree classifier means we cannot build the trees in parallel
  - The k-th classifer f_k can only be constructed after all $f_1, \ldots, f_{k-1}$ have been built.
- The construction of the **Decision Tree** itself has some parallelism opportinities, such as evaluating multiple split points in parallel [see Shafer et al. 1996]

Gradient Boosted Decision Tree Example

In [3]:

```
from itertools import islice
from sklearn.ensemble import GradientBoostingRegressor

plotdata()

# Boosted decision tree with K=100 trees, each with maximum depth 2, and lear
DT = GradientBoostingRegressor(n_estimators=100, max_depth=2, learning_rate=1
DT.fit(x_train, y)

ax = plt.gca()

# step through prediction as we add 10 more trees.
colorvector = ['g','b']
legendvector = ['f_1',"f_1 + f_2"]
i = 0
for pred in islice(DT.staged_predict(new_x[:, np.newaxis]), 0, 2, 1):
    plt.plot(new_x, pred, color=colorvector[i], alpha=1,label='Grad Boost Reg
    i += 1

pred = DT.predict(new_x[:, np.newaxis])
plt.plot(new_x, pred, color='r', label='Grad Boost Reg Tree f_1+...+f_'+str(D

plt.legend(loc='upper left')
plt.show()
```



Boosted Decision Trees for Loan Repayment Prediction

- In this example, we showcase the *boosted decision tree*'s ability to fit the training data
  - The ability to better fit the data explains the dominace of boosted decision trees for structured data, where the correct input features are clear
  - Neural networks dominate unstructured data, such as images and natural language text

In [4]:

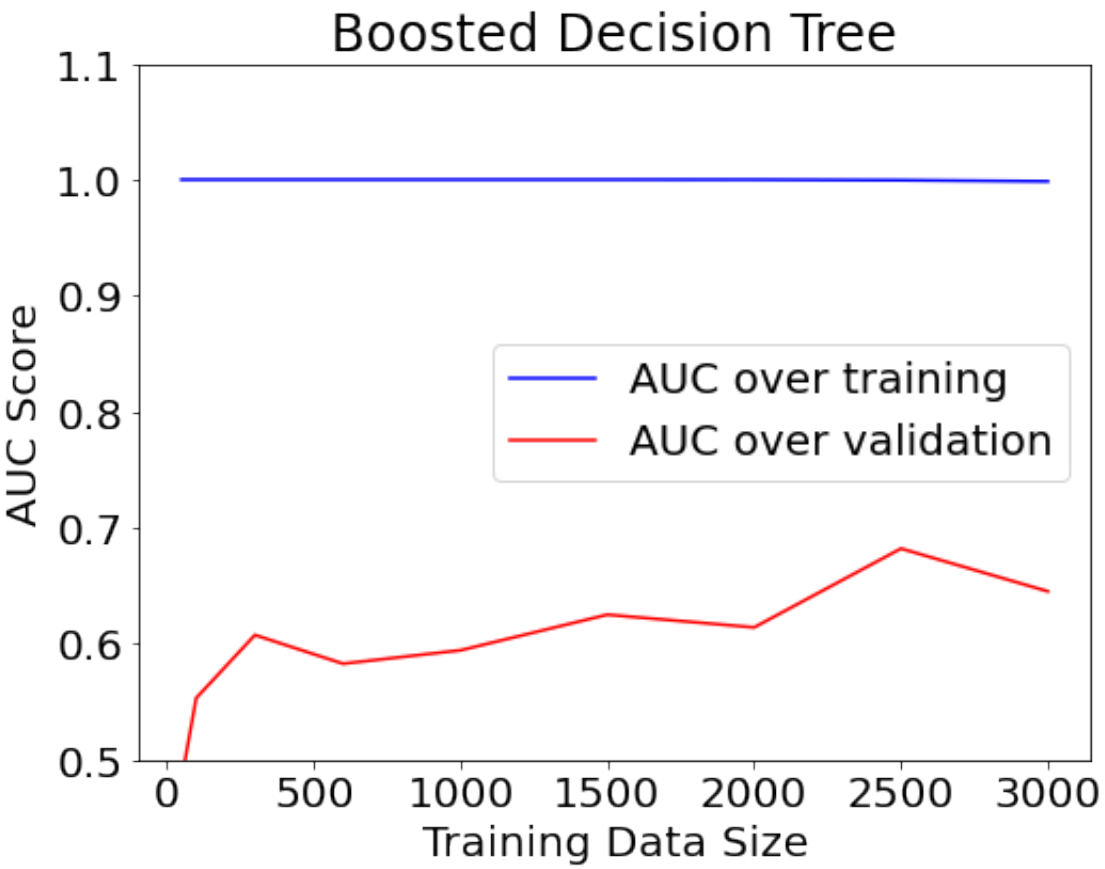
```
from statistical_header import *
from sklearn.ensemble import GradientBoostingRegressor


#Data at
#     https://www.cs.purdue.edu/homes/ribeirob/courses/Fall2020/lectures/Week
#     https://www.cs.purdue.edu/homes/ribeirob/courses/Fall2020/lectures/Week

train_file = "Bank_Data_Train.csv"
validation_file = "Bank_Data_Validation.csv"

train_sizes = [50, 100,300,600,1000,1500,2000, 2500, 3000]
AUC_scores_validation = []
AUC_scores_train = []


# Boosted decision tree with K=1000 trees, each with maximum depth 3, and lea
#    The default loss function is the squared error: l(y,y') = (y - y')^2
DT = GradientBoostingRegressor(n_estimators=500, max_depth=3, learning_rate=0

for train_size in train_sizes:

    # read the data
    print(f'Training data size {train_size}')

    # Process the data
    data_train, target_train = data_processing(train_file)
    data_validation, target_validation = data_processing(validation_file)

    # Get the first train_size examples
    data_train = data_train[0:train_size]
    target_train = target_train[0:train_size]

    # replace categorical strings with one-hot encoding and add a small amoun
    data_train, data_validation = add_categorical(train=data_train,validation
    data_train, data_validation = add_categorical(train=data_train,validation


    ## Fit the data
    DT.fit(data_train, target_train)
```

```
    ## Predictions
    y_pred_train = DT.predict(data_train)
    y_pred_val = DT.predict(data_validation)


    ## Performance Metrics

    ### ROC Curve ###
    # Find false and true positives of each target for various tresholds
    fpr, tpr, thresholds =roc_curve(target_train, y_pred_train)
    # Area under the curve
    roc_auc = auc(fpr, tpr)
    AUC_scores_train.append(roc_auc)

    # Find false and true positives of each target for various tresholds
    fpr, tpr, thresholds =roc_curve(target_validation, y_pred_val)
    # Area under the curve
    roc_auc = auc(fpr, tpr)
    AUC_scores_validation.append(roc_auc)


print(f'AUC_scores_validation={AUC_scores_validation}')
print(f'AUC_scores_train={AUC_scores_train}')
print("Done.")
```

```
Training data size 50
Training data size 100
Training data size 300
Training data size 600
Training data size 1000
Training data size 1500
Training data size 2000
Training data size 2500
Training data size 3000
AUC_scores_validation=[0.4828626410776512, 0.5533191172136406, 0.6076871066729
079, 0.582964927792514, 0.5945111908600753, 0.6251972053189092, 0.614240391116
6588, 0.682148367746745, 0.6453078136648116]
AUC_scores_train=[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.999964987360437, 0.999521103
4595793, 0.9985090992522819]
Done.
```

In [5]:

```
%matplotlib inline
#prepare plots
pl.rcParams["figure.figsize"] = (8,6)
pl.rcParams.update({'font.size': 20})
fig, ax = pl.subplots()
pl.plot(train_sizes, AUC_scores_train, label='AUC over training',color="blue"
pl.plot(train_sizes, AUC_scores_validation, label='AUC over validation',color
ax.set_ylim([0.5,1.1])
pl.legend(loc='best')
pl.xlabel('Training Data Size')
pl.ylabel('AUC Score')
pl.title('Boosted Decision Tree')

pl.show()
```



Decision Trees: Bias-Variance Trade-off

- The above plot shows that Boosted Decision Trees are complex models that can take advantage of having a lot of data.
  - Even though Boosted Decision Trees are built on top of weak classifiers (e.g., decision stumps for the above tree), the final model is surprisingly complex (perfectly fit the training data)
    - How do we know it is a complex model? Training error E_{train} remains nearly zero as training data increases (note that plot show AUC, not error)
  - Generalization gap E_{approx} is large but reduces as training data size increases
  - Boosted Decision Trees do really well with very large datasets

---

Extra: What about expanding the Taylor series?

Here we have removed the learning rate α to simplify exposition.

$$\text{Score}^{(k)} = \sum_{i=1}^{N} l\left(y_i, \hat{y}_i^{(k-1)} + f_k(x_i)\right) + \sum_{m=1}^{k} \Omega(f_m)$$

- Consider the Taylor expansion of the score:
  - Recall that $f(x+\Delta x) \approx f(x) + \frac{d}{dx} f(x)\Delta x + \frac{1}{2}\frac{d^2}{dx^2} f(x)\Delta x^2$
  - Define

$$h_i^{(k)} = \frac{\partial}{\partial \hat{y}^{(k-1)}} l(y_i, \hat{y}^{(k-1)}),$$

and

$$g_i^{(k)} = \frac{\partial^2}{\partial^2 \hat{y}^{(k-1)}} l(y_i, \hat{y}^{(k-1)}).$$

  - Then, the Taylor series approximation yields:

$$\begin{aligned}\text{Score}^{(k)} &\approx \sum_{i=1}^{N}\left(l(y_i, \hat{y}_i^{(k-1)}) + h_i^{(k)} f_k(x_i) + \frac{1}{2} g_i^{(k)} f_k(x_i)^2\right) + \Omega(f_k) \\ &\approx \sum_{i=1}^{N}\left(h_i^{(k)} f_k(x_i) + \frac{1}{2} g_i^{(k)} f_k(x_i)^2\right) + \Omega(f_k) + \text{constant}\end{aligned}$$

where in the last equation we used the fact that at the k-th iteration, $l(y_i, \hat{y}_i^{(k-1)})$ is

constant.

If we define the tree complexity as

$$\Omega(f_k) = \gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} w_j^2,$$

where w_j is the prediction given by leaf j, i.e., $w_{q^{(k)}(x_i)} = f(x_i)$.

The final score will be

$$\begin{aligned}\text{Score}^{(k)} &\approx \sum_{i=1}^{N}\left(h_i^{(k)} f_k(x_i) + \frac{1}{2} g_i^{(k)} f_k(x_i)^2\right) + \gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} w_j^2 + \text{constant} \\ &\approx \sum_{j=1}^{J}\left(\sum_{i\in I_j} h_i^{(k)}\right) w_j + \frac{1}{2}\left(\sum_{i\in I_j} g_i^{(k)} + \lambda\right) w_j^2 + \gamma J + \text{constant}\end{aligned}$$

References

- J.H. Friedman, Greedy function approximation a gradient boosting machine, 2001.
  - First paper on gradient boosting
- J.H. Friedman, Stochastic Gradient Boosting, 2002.
  - Presents a stochastic gradient version of gradient boosting.
- Elements of Statistical Learning. T. Hastie, R. Tibshirani and J.H. Friedman
  - Contains a chapter about gradient boosted boosting
- J.H. Friedman T. Hastie R. Tibshirani, Additive logistic regression a statistical view of boosting, 2000.
  - Uses second-order statistics for tree splitting, which is related to the expanded Taylor series.