1 Lab 13: Bagging, Boosting and Photometric Redshift

In this notebook, we explore the powerful ensemble class of methods in sklearn. We use Random Forests, AdaBoost and GradientBoosting to estimate photometric redshifts starting from observations of galaxy magnitudes in six different photometric bands (u, g, r, i, z, y).

Essentially, we try to reproduce/improve upon the results of this paper (https://arxiv.org/abs/1903.08174), for which the data are public and available here (http://d-scholarship.pitt.edu/36064/).

To get an idea of what we are shooting for, we can look at a figure in the paper:

Performance of photometric redshift reconstruction

In the figure above, σ_{NMAD} is the normalized median absolute deviation of the residual vector, and η is the fraction of outliers, defined as those objects for which (z_true - z_est)/(1+z_true) > 0.15. To be fair, we are working with DEEP2/3 data, so our range is slightly different.

In [5]:
```python
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt

pd.set_option('display.max_columns', 100)
pd.set_option('display.max_rows', 100)
pd.set_option('display.max_colwidth', 100)

font = {'size'   : 16}
matplotlib.rc('font', **font)
matplotlib.rc('xtick', labelsize=14)
matplotlib.rc('ytick', labelsize=14)
matplotlib.rcParams.update({'figure.autolayout': False})
matplotlib.rcParams['figure.dpi'] = 100
```

In [6]:
```python
from sklearn import metrics
from sklearn.model_selection import cross_validate, KFold, cross_val_predict, GridSearchCV
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, ExtraTreesRegressor
```

In [7]:
```python
import astropy
from astropy.io import fits
#fits stands for Flexible Image Transport System; it's a format that allows one to store images and summary data
```

1.1 Step 1: Data import and preparation

We can read the data into a data frame using pandas:

In [8]:
```python
with fits.open('DEEP2_uniq_Terapix_Subaru_v1.fits') as data:
    df = pd.DataFrame(np.array(data[1].data).byteswap().newbyteorder()) #see https://numpy.org/devdocs/user/basics.byteswapping.html#changin
```

1. How many features and examples are in the dataset?

In [11]:
```python
df
print("""
There are 23822 examples and 78 features in the dataset
""")
```

```
There are 23822 examples and 78 features in the dataset
```

2. What are the names of the features?

In [13]:
```python
df.columns.values
```

Out[13]:
```
array(['objno_deep2', 'ra_deep2', 'dec_deep2', 'magb', 'magr', 'magi',
       'pgal', 'sfd_ebv', 'class', 'subclass', 'objname', 'maskname',
       'slitname', 'date', 'mjd', 'z_raw', 'zhelio', 'z_err', 'rchi2',
       'dof', 'vdisp', 'vdisp_err', 'zquality', 'egsflags', 'comment',
       'm_b', 'ub_0', 'ra_cfhtls', 'dec_cfhtls', 'u', 'g', 'r', 'i', 'i2',
       'z', 'uerr', 'gerr', 'rerr', 'ierr', 'i2err', 'zerr', 'u_apercor',
       'g_apercor', 'r_apercor', 'i_apercor', 'i2_apercor', 'z_apercor',
       'uerr_aper', 'gerr_aper', 'rerr_aper', 'ierr_aper', 'i2err_aper',
       'zerr_aper', 'uerr_apercor', 'gerr_apercor', 'rerr_apercor',
       'ierr_apercor', 'i2err_apercor', 'zerr_apercor', 'r_radius_arcsec',
       'u(sexflag)', 'g(sexflag)', 'r(sexflag)', 'i(sexflag)',
       'i2(sexflag)', 'z(sexflag)', 'flag_cfhtls', 'cfhtls_source',
       'ra_subaru', 'dec_subaru', 'y', 'yerr', 'y_apercor', 'yerr_aper',
       'yerr_apercor', 'y(sexflag)', 'y_radius_arcsec', 'subaru_source'],
      dtype=object)
```

3. Define a dataframe that contains as features the six galaxy brightness bands of interest: *u_apercor, g_apercor, r_apercor, i_apercor, z_apercor,y_apercor*

In [14]:
```python
features = df[['u_apercor', 'g_apercor', 'r_apercor', 'i_apercor', 'z_apercor','y_apercor']]
```

4. Define a dataframe that contains as target the spectroscopic redshifts, found under the label *zhelio*

In [15]:
```python
target = df['zhelio']
```

1.2 Step 2 : The first Random Forest model!

1. Define a RandomForestRegressor() model. Check out what hyperparameters it has by calling .get_params().
2. Establish a benchmark using our usual 3-fold cross-validated scores with shuffling, It takes a little time, to speed things up try setting n_jobs = -1, which uses all cores on your CPU. What is the default evaluation metric here?

3. Does the model suffer from high bias or high variance?

In [19]:
```
model = RandomForestRegressor(n_estimators=100, random_state=42)
params = model.get_params()
print(params)
```

```
{'bootstrap': True, 'ccp_alpha': 0.0, 'criterion': 'squared_error', 'max_depth': None, 'max_features': 1.0, 'max_leaf_nodes': None, 'max_sam
ples': None, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 1
00, 'n_jobs': None, 'oob_score': False, 'random_state': 42, 'verbose': 0, 'warm_start': False}
```

In [21]:
```
n_splits = 3
kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
```

In [22]:
```
scores = cross_validate(model, features, target, cv=kf, n_jobs=-1 , return_train_score=True )
```

In [26]:
```
print(f"test scores {scores['test_score']}")
print(f"train scores {scores['test_score']}")
print(f"test scores mean {scores['test_score'].mean()}")
print(f"train scores mean {scores['train_score'].mean()}")
```

```
test scores [0.3217413  0.28238929 0.31415639]
train scores [0.3217413  0.28238929 0.31415639]
test scores mean 0.306095660325408l4
train scores mean 0.876322085220091
```

In [27]:
```
print('High Variance')
```

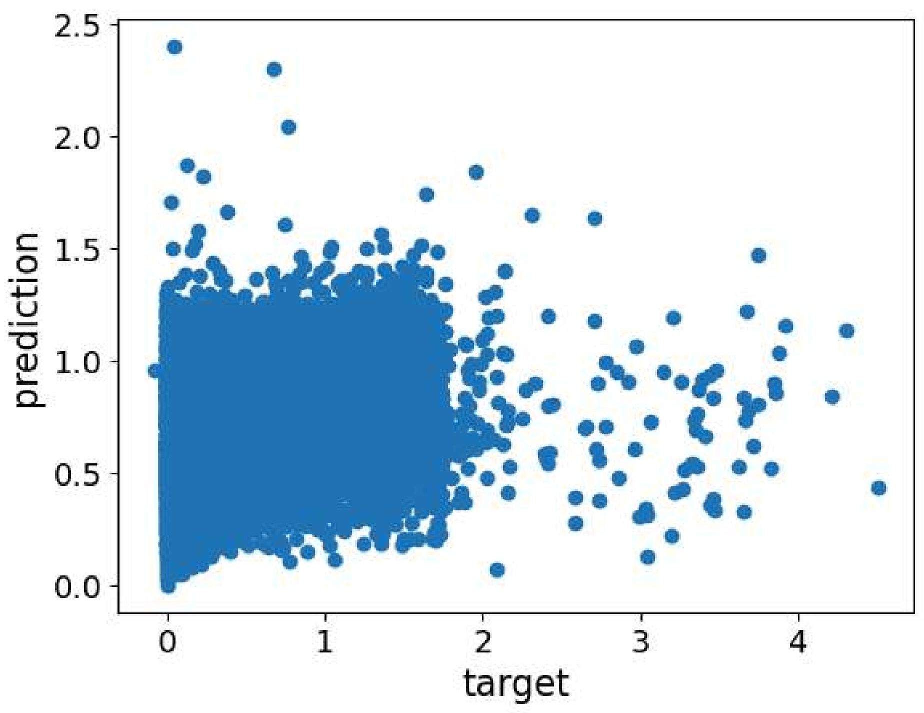
```
High Variance
```

4. Generate cross-validated predictions and plot target (i.e. truth) against predictions. Does the plot look like the one from the beginning of the notebook?

In [29]:
```
prediction = cross_val_predict(model, features, target, cv = kf, n_jobs= -1)
```

In [31]:
```
plt.scatter(target,prediction)
plt.ylabel('prediction')
plt.xlabel('target')
```

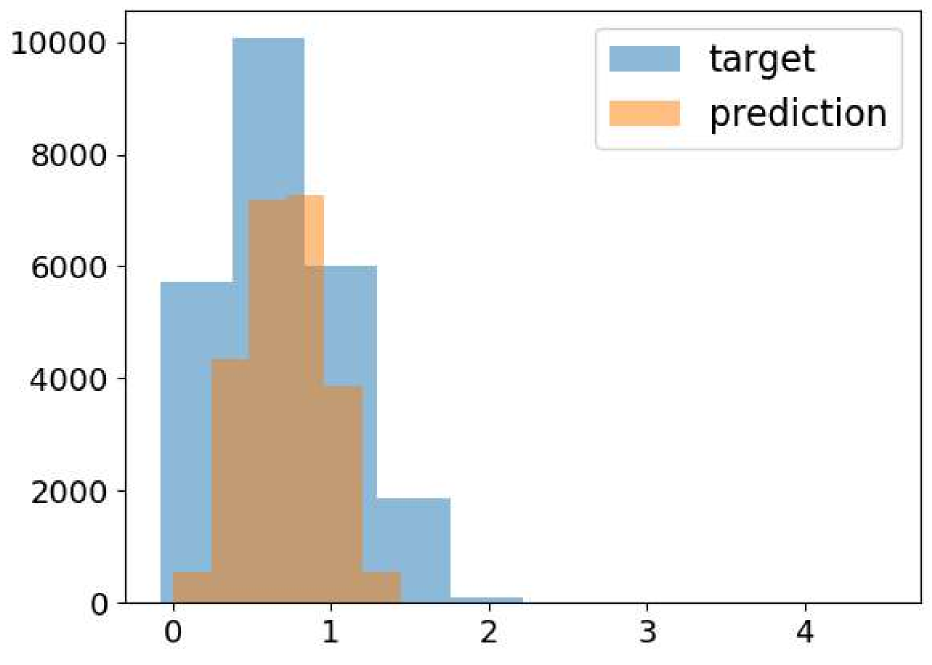
Out[31]:
```
Text(0.5, 0, 'target')
```



4. It's also interesting to look at the distribution of the predicted values. Plot histograms of both predicted and true values. Any observations?

In [36]:
```
plt.hist(target,alpha = 0.5,label='target')
plt.hist(prediction,alpha = 0.5,label='prediction')
plt.legend()
```

Out[36]:
```
<matplotlib.legend.Legend at 0x19bd937f730>
```



5. Calculate the outlier fraction η defined at the beginning, does it compare to the data from the pape?

```
In [38]: sum((target - prediction)/(1+target) > 0.15)/target.shape[0]
```

```
Out[38]: 0.1013768785156578
```

```
In [39]: print("""
         The paper has a reading of 5% \n
         ------------------------------\n
         Effectively there is a difference of 5%
         """)
```

```
The paper has a reading of 5%

------------------------------

Effectively there is a difference of 5%
```

1.3 Step 3: Parameter Optimization.

1. We can start by making the data set a bit smaller, as we have seen that timings were already challenging in simple k-fold CV. Generate a smaller sample of 5000 examples. They should be drawn randomly without replacement from the full dataset, try *np.random.choice*

```
In [42]: df_random = df.sample(n=5000, random_state=42)
```

```
In [43]: features_subset = df_random[['u_apercor', 'g_apercor', 'r_apercor', 'i_apercor', 'z_apercor','y_apercor']]
         target_subset = df_random['zhelio']
```

2. It is good practice to ensure that the performance on the smaller set remains similar to the one obtained on the entire data set, which means that the change in size will not significantly affect the optimization process. So run the 3-fold crossvalidation again on the smaller dataset

```
In [50]: model = RandomForestRegressor()
         params = model.get_params()

         kf = KFold()

         n_splits = 3
         kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)

         scores = cross_validate(model, features_subset, target_subset, cv=kf, n_jobs=-1 , return_train_score=True )

         print(f"test scores {scores['test_score']}")
         print(f"train scores {scores['test_score']}")
         print(f"test scores mean {scores['test_score'].mean()}")
         print(f"train scores mean {scores['train_score'].mean()}")
```

```
test scores [0.30619419 0.24871713 0.27992573]
train scores [0.30619419 0.24871713 0.27992573]
test scores mean 0.2782790183511093
train scores mean 0.8611766088669718
```

3. We are now ready to optimize hyperparameters. Here are some options:

1.3.0.1 Tree Parameters

Some useful parameters associated to a tree are:

- The minimum number of instances in a leaf node;
- The minimum number of instances required in a split node;
- The maximum depth of tree;
- The criterion chosen to decide whether a split is "worth it", expressed in terms of information gain.

1.3.0.2 Randomization Parameters

Here we find:

- The number of k < n features that are used in building trees;
- The re-sampling (boostrap) of the data set (T or F).

1.3.0.3 Forest Parameters

The number of trees in the forest (n_estimators) can be adjusted, with the general understanding that more trees are better, but at some point performance will plateau, so one can find the trade-off between having more trees and lower runtime.

Try running grid search to look for the best model within the following parameter subset. Are you able to improve the model significantly?

- min_impurity_decrease: 0, 0.1, 0.5
- max_leaf_nodes: None, 100, 200
- min_samples_split: 10, 20, 100
- max_features: None, 2, 4

In [52]:
```
from sklearn.model_selection import GridSearchCV

parameters = {
    'min_impurity_decrease': [0, 0.1, 0.5],
}

grid_search = GridSearchCV(model, parameters, cv=kf, scoring='accuracy', n_jobs=-1)
grid_search.fit(features_subset, target_subset)

best_model = grid_search.best_estimator_  # Access the model with the best hyperparameters
best_score = grid_search.best_score_  # Get the best score achieved

print("Best parameters:", best_model.get_params())
print("Best score:", best_score)
```

```
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\model_selection\_search.py:976: UserWarning: One or more of the test scores a
re non-finite: [nan nan nan]
  warnings.warn(

Best parameters: {'bootstrap': True, 'ccp_alpha': 0.0, 'criterion': 'squared_error', 'max_depth': None, 'max_features': 1.0, 'max_leaf_node
s': None, 'max_samples': None, 'min_impurity_decrease': 0, 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0,
'n_estimators': 100, 'n_jobs': None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
Best score: nan
```

In [48]:
```
print("We find that the optimization doesn't seem to improve the outcome")
```

Out[48]:
```
0
```

4. In addition to hyperparameter optimization, another possibility for improving the model could be to consider additional features that might help the decision trees.

We need to retain some additional columns to be used in the selection process. Try adding *'subaru_source','cfhtls_source','zquality'* to the already existing six features.

In [65]:
```
mags = df[['u_apercor', 'g_apercor', 'r_apercor', 'i_apercor', 'z_apercor','y_apercor','subaru_source','cfhtls_source','zquality','zhelio']]
mags.shape
```

Out[65]:
```
(23822, 10)
```

There are a few more clean-up steps that help in the process. Finding these constitutes many hours of research work, but here we take a shortcut and benefit from the knowledge of the authors of the redshift paper.

a) only use objects with high-quality spectroscopic redshift measurements, zquality>3

b) select objects with cfhtls deep photometric data, cfhtls_source = 0

In [76]:
```
filtered_df = mags.query('zquality >= 3 and cfhtls_source == 0')
```

c) Unavailable measurements are marked by -99 or 99 (while typical values are around 20-25). We also get rid of data with missing measurements.

In [78]:
```
filtered_df = filtered_df[filtered_df > -10].dropna()
filtered_df = filtered_df[filtered_df < 90].dropna()
```

Now that we have curated this extended data set with the additional information, we only keep the six original features in this dataset in our final dataset that should have 6,307 objects. We need, of course, to select the same set on the target vector.

In [79]:
```
mags_subset = filtered_df[['u_apercor', 'g_apercor', 'r_apercor', 'i_apercor', 'z_apercor','y_apercor']]

mags_target = filtered_df['zhelio']
```

5. Evaluate this new benchmark model via cross-validation. Note that for reproducible results we need to fix the random_state=5 parameter of the Random Forest (which controls the bootstrap process) and the random_state=10 of the cross validation.

Find the new mean test and train scores, has the model improved?

In [80]:
```
model = RandomForestRegressor()
params = model.get_params()

kf = KFold()

n_splits = 3
kf = KFold(n_splits=n_splits, shuffle=True, random_state=10)

scores = cross_validate(model, mags_subset, mags_target, cv=kf, n_jobs=-1 , return_train_score=True )

print(f"test scores {scores['test_score']}")
print(f"train scores {scores['test_score']}")
print(f"test scores mean {scores['test_score'].mean()}")
print(f"train scores mean {scores['train_score'].mean()}")
```

```
test scores [0.7229522  0.66783251 0.83397507]
train scores [0.7229522  0.66783251 0.83397507]
test scores mean 0.7415865913167131
train scores mean 0.9637139256482321
```

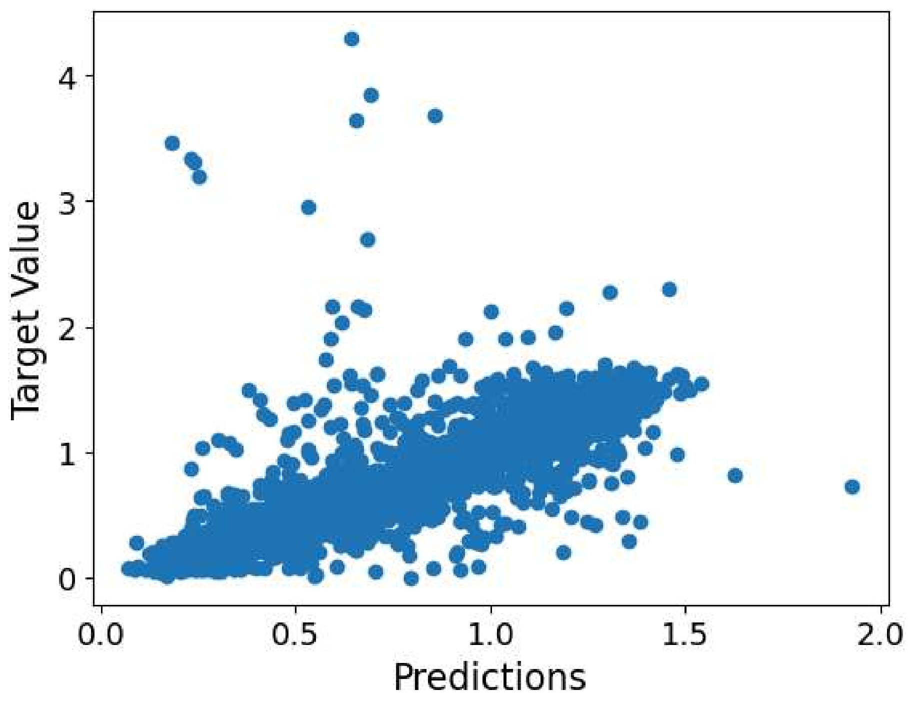
We can re-run the optimization process (note that the data set size is limited, so we don't need to make it smaller), but let's skip this for now in the interest of time.

Finally, we can generate one set of predictions to visualize what happens. Plot again observed values against predictions. Any improvement? Also compute the outlier fraction.

In [84]:
```
prediction = cross_val_predict(model, mags_subset, mags_target, cv=kf, n_jobs=-1 )
```

```
In [86]: plt.scatter(prediction,mags_target)
         plt.xlabel('Predictions')
         plt.ylabel('Target Value')
```

```
Out[86]: Text(0, 0.5, 'Target Value')
```



1.4 Step 4: Boosting Methods

Now that we have explored random forests, let's take a look at the boosting strategy.

```
In [87]: from sklearn.ensemble import AdaBoostRegressor
```

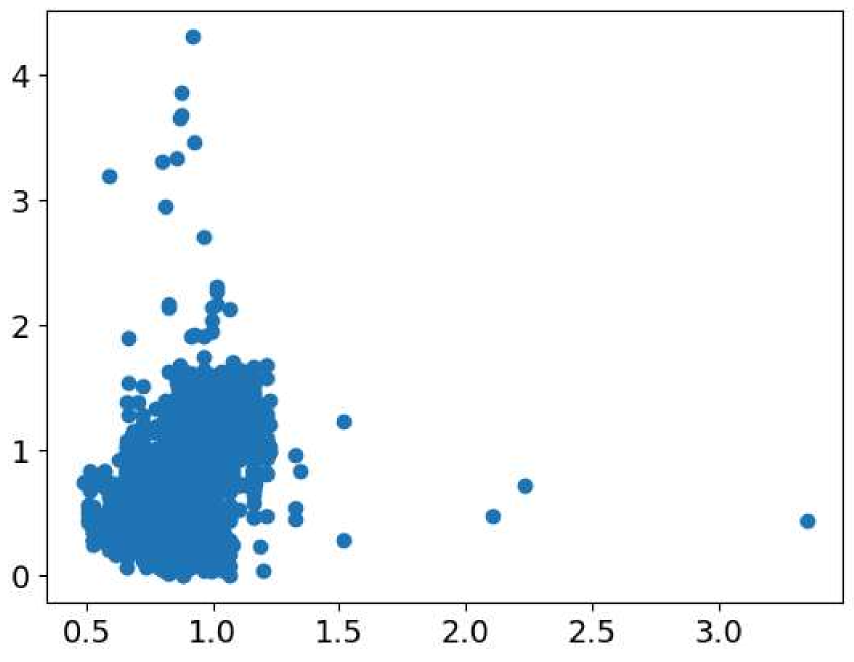
1. First, set up the AdaBoostRegressor with the default parameters, compute cross-validated prediction and make a scatter plot of predictions versus true values. Also check out the default parameters via .get_params(). Any comments?

```
In [88]: model = AdaBoostRegressor()
         params = model.get_params()

         kf = KFold()

         n_splits = 3
         kf = KFold(n_splits=n_splits, shuffle=True, random_state=10)

         scores = cross_validate(model, mags_subset, mags_target, cv=kf, n_jobs=-1 , return_train_score=True )

         print(f"test scores {scores['test_score']}")
         print(f"train scores {scores['test_score']}")
         print(f"test scores mean {scores['test_score'].mean()}")
         print(f"train scores mean {scores['train_score'].mean()}")
```

```
test scores [-0.17722003  0.02236971  0.00974877]
train scores [-0.17722003  0.02236971  0.00974877]
test scores mean -0.048367182947866426
train scores mean 0.05192761550516326
```

```
In [89]: prediction = cross_val_predict(model, mags_subset, mags_target, cv=kf, n_jobs=-1 )

         plt.scatter(prediction,mags_target)
```

```
Out[89]: <matplotlib.collections.PathCollection at 0x19be0213730>
```



2. The AdaBoostRegressor has a nice property called "staged_predict" that lets one examine how the prediction gradually improves with stacking at each stage. Create a train/test split (test_size=0.3) because we need to use the ".fit" method in order to access the "staged_predict" property later.

Initialize AdaBoostRegressor with **estimator=DecisionTreeRegressor(max_depth=3),n_estimators=30**, fit the AdaBoostRegressor to the training set, then use the staged_predict property to make a plot of R2-score versus iteration for up to the 30 estimators.

In [90]:
```python
model = AdaBoostRegressor(estimator=DecisionTreeRegressor(max_depth=3),n_estimators=3)
params = model.get_params()

kf = KFold()

n_splits = 3
kf = KFold(n_splits=n_splits, shuffle=True, random_state=10)

scores = cross_validate(model, mags_subset, mags_target, cv=kf, n_jobs=-1 , return_train_score=True )

print(f"test scores {scores['test_score']}")
print(f"train scores {scores['test_score']}")
print(f"test scores mean {scores['test_score'].mean()}")
print(f"train scores mean {scores['train_score'].mean()}")
```

```
test scores [0.2101863  0.19793679 0.23052874]
train scores [0.2101863  0.19793679 0.23052874]
test scores mean 0.21288394567414837
train scores mean 0.2641374878504613
```

In [ ]:

3. The default estimator in AdaBoost is a decision tree with max_depth=3. Repeat the above calculation with a stronger base learner, i.e. max_depth=6 and max_depth=10 and combine all three curves in one graph. Any comments?

In [ ]:

4. Now try out GradientBoostingRegressor and produce the same plot with this alternative method. Any comments?

In [ ]: