Primo Modello

In questo notebook sono usati i dataset su cui è stata operata la fase preliminare di **preparazione dei dati**, la **divisione** in base alla regione e la **selezione delle feature**.

In prima instanza è allenato un **albero di decisione**; in seconda istanza si tenta di diminuire l'errore dell'albero sfruttando un algoritmo di **boosting**.

Per entrambi i modelli si prevede un algoritmo per fare **tuning degli iperparametri**: rispettivamente il **massimo numero di foglie** per l'albero di decisione e il **numero di modelli impiegati** per il modello di boosting. Per fare tuning dei parametri si fa uso di un **dataset di validation**: si mantiene l'iperparametro che produce il minimo errore su validation.

In [1]:
```
# Libraries
import pandas            as pd
import numpy             as np
import matplotlib.pyplot as plt
import warnings

from sklearn.metrics         import mean_squared_error
from sklearn.tree            import DecisionTreeRegressor
from sklearn.tree            import plot_tree
from sklearn.preprocessing   import StandardScaler
from sklearn.utils           import resample
from sklearn.ensemble        import AdaBoostRegressor

warnings.filterwarnings('ignore')
```

Lettura dei dati

In [2]:
```
# local file paths

dir_name = 'selezione'
region_names = np.array(['A', 'B', 'C'])

fp_Xtrain = []
fp_Xval   = []
fp_Xtest  = []
fp_ytrain = []
fp_yval   = []
fp_ytest  = []

for i in range(3):
    fp_Xtrain.append(dir_name + f'/X_train{region_names[i]}.csv')
    fp_Xval  .append(dir_name + f'/X_val{  region_names[i]}.csv')
    fp_Xtest .append(dir_name + f'/X_test{ region_names[i]}.csv')
    fp_ytrain.append(dir_name + f'/y_train{region_names[i]}.csv')
    fp_yval  .append(dir_name + f'/y_val{  region_names[i]}.csv')
    fp_ytest .append(dir_name + f'/y_test{ region_names[i]}.csv')
```

Leggo i dataset su cui è stata fatta la selezione delle feature

In [3]:
```
# Lettura dei dati

X_train = []
X_val   = []
X_test  = []
y_train = []
y_val   = []
y_test  = []

for i in range(3):
    X_train.append(pd.read_csv(fp_Xtrain[i], low_memory=False))
    X_val  .append(pd.read_csv(fp_Xval  [i], low_memory=False))
    X_test .append(pd.read_csv(fp_Xtest [i], low_memory=False))
    y_train.append(pd.read_csv(fp_ytrain[i], low_memory=False))
    y_val  .append(pd.read_csv(fp_yval  [i], low_memory=False))
    y_test .append(pd.read_csv(fp_ytest [i], low_memory=False))

X_train = np.array(X_train, dtype=object)
X_val   = np.array(X_val,   dtype=object)
X_test  = np.array(X_test,  dtype=object)
y_train = np.array(y_train, dtype=object)
y_val   = np.array(y_val,   dtype=object)
y_test  = np.array(y_test,  dtype=object)
```

In [4]:
```
def dimensionality(y=False):
    for i in range(3):
        print(f'X_train{region_names[i]}: {X_train[i].shape}')
        print(f'X_val{region_names[i]}:   {X_val  [i].shape}')
        print(f'X_test{region_names[i]}:  {X_test [i].shape}')
        if y:
            print(f'y_train{region_names[i]}: {y_train[i].shape}')
            print(f'y_val{region_names[i]}:   {y_val  [i].shape}')
            print(f'y_test{region_names[i]}:  {y_test [i].shape}')
        print()
```

In [5]:
```
dimensionality(y=True)
```

```
X_trainA: (26819, 55)
X_valA:   (9006, 55)
X_testA:  (9085, 55)
y_trainA: (26819, 1)
y_valA:   (9006, 1)
y_testA:  (9085, 1)

X_trainB: (8119, 32)
X_valB:   (2658, 32)
X_testB:  (2606, 32)
y_trainB: (8119, 1)
y_valB:   (2658, 1)
y_testB:  (2606, 1)

X_trainC: (64771, 33)
X_valC:   (21908, 33)
X_testC:  (21876, 33)
y_trainC: (64771, 1)
y_valC:   (21908, 1)
y_testC:  (21876, 1)
```

Variabili globali

In [6]:
```
# Nomi delle regioni
region_ids   = np.array(['1286', '2061', '3101'])

# Riduzione del dataset C
C_PERC = 2/5
C_IND  = 2

# Percentuale per il sottoinsieme su cui costruire il modello
#SUB_PERC = [1/1000, 1/100, 1/1000] # A:   100, B:  100, C:   100; Test rapido      ~   5 min
SUB_PERC = [   1/3,     1,    1/3]  # A:  8000, B: 8000, C:  8000; Test medio       ~   3 ore
#SUB_PERC = [     1,     1,      1] # A: 24000, B: 8000, C: 64000; Dataset Completo ~  21 ore

# Decision Tree Regressor
DTR_START =  10
DTR_END   = 300
DTR_STEP  =  10

# AdaBoost Regressor
BOOST_START =  20
BOOST_END   = 500
BOOST_STEP  =  20
BOOST_NTEST =  20
#BOOST_SAMPLE_PERC = 1/3 # 0.33
BOOST_SAMPLE_PERC  = 2/3 # 0.67
```

Limite del numero di instanze del terzo dataset

Per limiti tecnici di memoria, la gestione del dataset C risulta complicata in quato incorre spesso in errori a run-time di cateoria **MemoryError**: la macchina su cui è eseguito il notebook non è in grado di allocare la quantità di memoria richiesta per lavorare sull'intero dataset di Train (circa 64000 righe per 48 colonne).

È dunque inevitabile dover ridurre l'insieme con un numero di righe il cui calcolo è supportato, cioè circa il 40% del dataset. Il dataset per la terza regione (`regionidcounty` : 3101) è riscalato su 2/5 delle osservazioni per Train, Validation e Test.

L'operazione introduce dunque una forte approssimazione per questa regione, che per limiti tecnici è però inevitabile.

In [7]:
```
for X, y in zip([X_train, X_val, X_test], [y_train, y_val, y_test]):
    X[C_IND], y[C_IND] = resample(
        X[C_IND], y[C_IND],
        n_samples = int(C_PERC * len(X[2]))
    )
```

In [8]:
```
dimensionality(y=True)
```

```
X_trainA: (26819, 55)
X_valA:   (9006, 55)
X_testA:  (9085, 55)
y_trainA: (26819, 1)
y_valA:   (9006, 1)
y_testA:  (9085, 1)

X_trainB: (8119, 32)
X_valB:   (2658, 32)
X_testB:  (2606, 32)
y_trainB: (8119, 1)
y_valB:   (2658, 1)
y_testB:  (2606, 1)

X_trainC: (25908, 33)
X_valC:   (8763, 33)
X_testC:  (8750, 33)
y_trainC: (25908, 1)
y_valC:   (8763, 1)
y_testC:  (8750, 1)
```

Definizione di un Subset per la costruzione del modello

Gli algoritmi usati in questo notebook hanno un costo computazionale elevato, per questo è definito un sottoinsieme dei dataset originali su cui far girare gli algoritmi; questo per facilitare la fase di creazione e di testing e ottenere risultati verosimili in tempi utili a verificare il corretto funzionamento del processo.

Nella versione finale gli algoritmi useranno circa 8000 instanze delle instanze del dataset originale, poiché il dataset allenato nella sua totalità richiede per il boosting tempi non sostenibili.

In [9]:
```
for i in range(3):
    print(f'Region {region_names[i]}')
    print(int(len(X_train[i])*SUB_PERC[i]))
    print(int(len(X_val  [i])*SUB_PERC[i]))
    print()
```

```
Region A
8939
3002

Region B
8119
2658

Region C
8636
2921
```

In [10]:
```
X_train_sub = []
y_train_sub = []
X_val_sub   = []
y_val_sub   = []
```

In [11]:
```
for i in range(3):
    Xt_sub, yt_sub = resample(X_train[i], y_train[i], n_samples = int(SUB_PERC[i]*len(X_train[i])))
    Xv_sub, yv_sub = resample(X_val  [i], y_val  [i], n_samples = int(SUB_PERC[i]*len(X_val[i])))
    X_train_sub.append(Xt_sub)
    y_train_sub.append(yt_sub)
    X_val_sub  .append(Xv_sub)
    y_val_sub  .append(yv_sub)
```

In [12]:
```
def dimensionality_sub(y=False):
    for i in range(3):
        print(f'X_train_sub{region_names[i]}: {X_train_sub[i].shape}')
        print(f'X_val_sub{region_names[i]}:   {X_val_sub  [i].shape}')
        if y:
            print(f'y_train_sub{region_names[i]}: {y_train_sub[i].shape}')
            print(f'y_val_sub{region_names[i]}:   {y_val_sub  [i].shape}')
        print()
```

In [13]:
```
dimensionality_sub(y=True)
```

```
X_train_subA: (8939, 55)
X_val_subA:   (3002, 55)
y_train_subA: (8939, 1)
y_val_subA:   (3002, 1)

X_train_subB: (8119, 32)
X_val_subB:   (2658, 32)
y_train_subB: (8119, 1)
y_val_subB:   (2658, 1)

X_train_subC: (8636, 33)
X_val_subC:   (2921, 33)
y_train_subC: (8636, 1)
y_val_subC:   (2921, 1)
```

DecisionTreeRegressor

Tuning del massimo numero di foglie attraverso il parametro `max_leaf_nodes` . Per studiare l'andamento la funzione produce due grafici che illustrano l'andamento di $bias^2$, *varianza* e *mean squared error* in funzione del massimo numero di foglie sia per l'insieme di *Train* che per quello di *Validation*.

In [14]:
```
plt.rcParams.update({'font.size': 35})
```

In [15]:
```
def get_bias_var_mse(X, y, model):
        y_pred = model.predict(X)
        return {
            'bias': ((y - np.mean(y_pred))**2).mean(),\
            'var':  np.var(y_pred).mean(),\
            'mse':   ((y_pred - y.reshape(-1,1))**2).mean()
        }
```

In [16]:
```
# Costruzione Decision TreeRegressor
def DecisionTreeRegressor_validation(X_train, y_train, X_val, y_val, verbose=False, debug=False, file_name=''):

    def get_dec_tree_reg(max_leaf):
        dt = DecisionTreeRegressor(
            max_leaf_nodes = max_leaf#,
            #criterion = 'absolute_error'
        )
        dt.fit(X_train, y_train)
        return dt

    def bias_var_mse(X, y, model):
        stats = get_bias_var_mse(X, y, model)
        return stats['bias'],\
               stats['var'],\
               stats['mse']

    def plot_mse(stats, name):

        print (f'{name}: TUNING DEL MASSIMO NUMERO DI FOGLIE')
        print()

        for n in ['mse', 'bias', 'var']:

            min_  = min(stats[n])
            best  = (np.argmin(stats[n]) * DTR_STEP) + DTR_START

            print (f'Punteggio finale: {stats[n][-1]} ({DTR_END}) MaxLeaves')
            print (f'Best {n}: {min_}')
            print (f'Best number of MaxLeaves: {best}')
            print()

        fig, ax = plt.subplots(figsize=(len(stats['mse'])/2, 10))

        ax.tick_params(axis='both', which='major', labelsize=25)
        ax.tick_params(axis='both', which='minor', labelsize=15)

        ax.plot(range(DTR_START, DTR_END+1, DTR_STEP),  stats['mse' ], 'o-', label='MSE')
        ax.plot(range(DTR_START, DTR_END+1, DTR_STEP),  stats['bias'], 'o-', label='BIAS')
        ax.plot(range(DTR_START, DTR_END+1, DTR_STEP),  stats['var' ], 'o-', label='VARIANCE')

        ax.set_title(f"{name} MSE, BIAS, VARIANCE on different MaxLeaves", fontsize=15)
        ax.set_xlabel("Number of Max Leaves used", fontsize=15)
        ax.grid()
        ax.legend(prop={'size': 12})

        if file_name != '':
            fig.savefig('images/' + file_name + '_DecisionTreeRegressor_' + name + '.jpg')

    y_train = y_train.values.ravel()
    y_val   = y_val  .values.ravel()

    first = True

    info = []

    train_stats  = {
        'bias' : [],
        'var'  : [],
        'mse'  : []
    }

    val_stats    = {
        'bias' : [],
        'var'  : [],
        'mse'  : []
    }

    for max_leaf in range(DTR_START, DTR_END+1, DTR_STEP):

        if debug:
            print(f'{max_leaf}/{DTR_END}')

        model = get_dec_tree_reg(max_leaf)

        trn_bias, trn_var, trn_mse = bias_var_mse(X_train, y_train, model)
        val_bias, val_var, val_mse = bias_var_mse(  X_val,   y_val, model)

        train_stats['bias'].append(trn_bias)
        train_stats['var' ].append(trn_var )
        train_stats['mse' ].append(trn_mse )

        val_stats['bias'].append(val_bias)
        val_stats['var' ].append(val_var )
        val_stats['mse' ].append(val_mse )

        info.append(f'Max Leaf: {max_leaf}' +\
                    f'\n(Train MSE:       {trn_mse }  - Val MSE:      {val_mse })' +\
                    f'\n(Train Bias:      {trn_bias}  - Val Bias:     {val_bias})' +\
                    f'\n(Train Variance:  {trn_var }  - Val Variance: {val_var })' )

        if debug:
            print(f'{max_leaf}/{DTR_END}')

        if(first or val_mse < best_mse):
            first = False
            best_mse = val_mse
            best_max_leaf = max_leaf
            best_model = model

    if verbose:
        print()
        print(f'{name} MSE, BIAS, VARIANCE Train e Validation')
        print(*info, sep='\n')
        print()

    plot_mse(train_stats, "Train"     )
    plot_mse(  val_stats, "Validation")

    return best_model
```

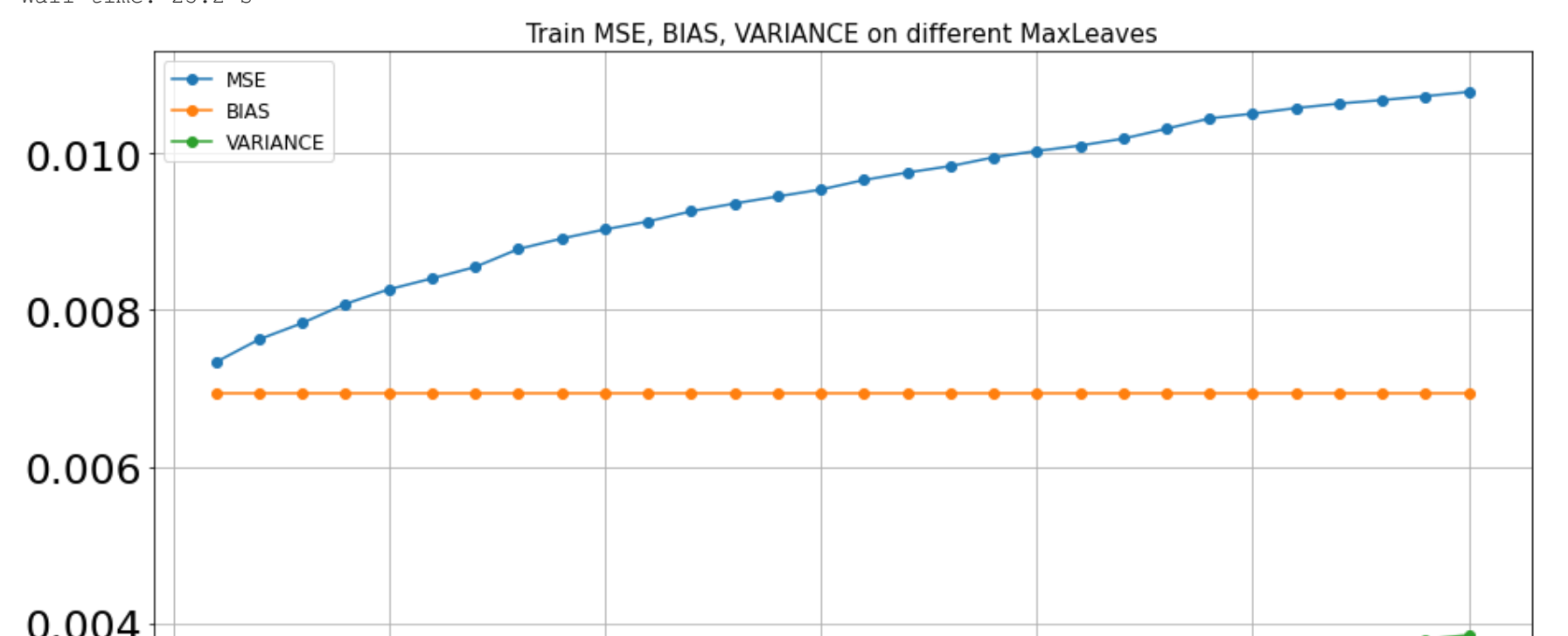
In [17]:
```
dt_model = []
```

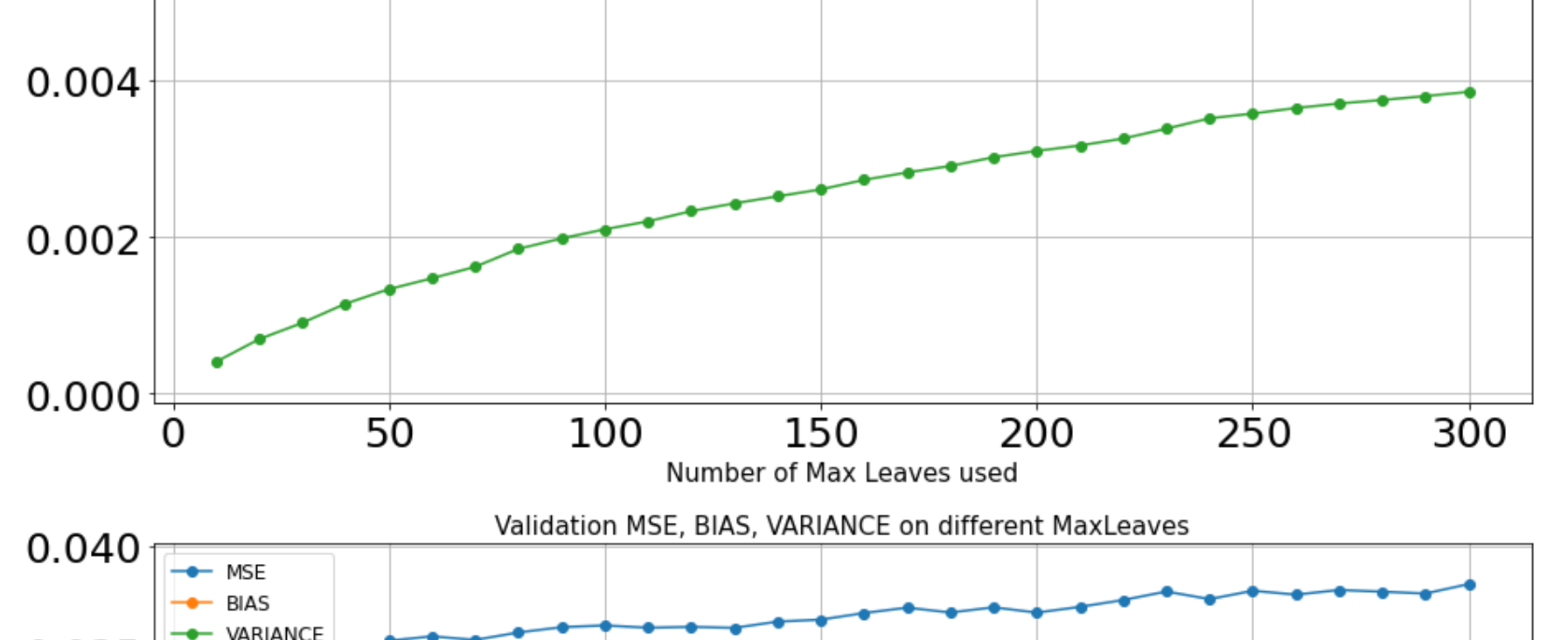
In [18]:
```
def get_dt(index, verbose=False, debug=False, file_name=''):
    if file_name == '':
        file_name = region_ids[index]
    return DecisionTreeRegressor_validation(
        X_train_sub[index],
        y_train_sub[index],
        X_val_sub  [index],
        y_val_sub  [index],
        verbose    = verbose,
        debug      = debug,
        file_name  = file_name
    )
```

Prima Regione 1286

In [19]:
```
%%time
dt_model.append(
    get_dt(
        0,
        verbose    = False,
        debug      = False#,
        #file_name = 'prova0'
    )
)
```

```
Train: TUNING DEL MASSIMO NUMERO DI FOGLIE

Punteggio finale: 0.01078387673563775 (300) MaxLeaves
Best mse: 0.007336282203794711
Best number of MaxLeaves: 10

Punteggio finale: 0.0069320060368185344 (300) MaxLeaves
Best bias: 0.0069320060368185344
Best number of MaxLeaves: 10

Punteggio finale: 0.0038518706988191907 (300) MaxLeaves
Best var: 0.00040427616697616165
Best number of MaxLeaves: 10

Validation: TUNING DEL MASSIMO NUMERO DI FOGLIE

Punteggio finale: 0.038210018282921705 (300) MaxLeaves
Best mse: 0.033863314023928795
Best number of MaxLeaves: 10

Punteggio finale: 0.03364000821357107 (300) MaxLeaves
Best bias: 0.03361805250079998
Best number of MaxLeaves: 50

Punteggio finale: 0.0045700100693506504 (300) MaxLeaves
Best var: 0.00023888553095341306
Best number of MaxLeaves: 10

Wall time: 25.2 s
```

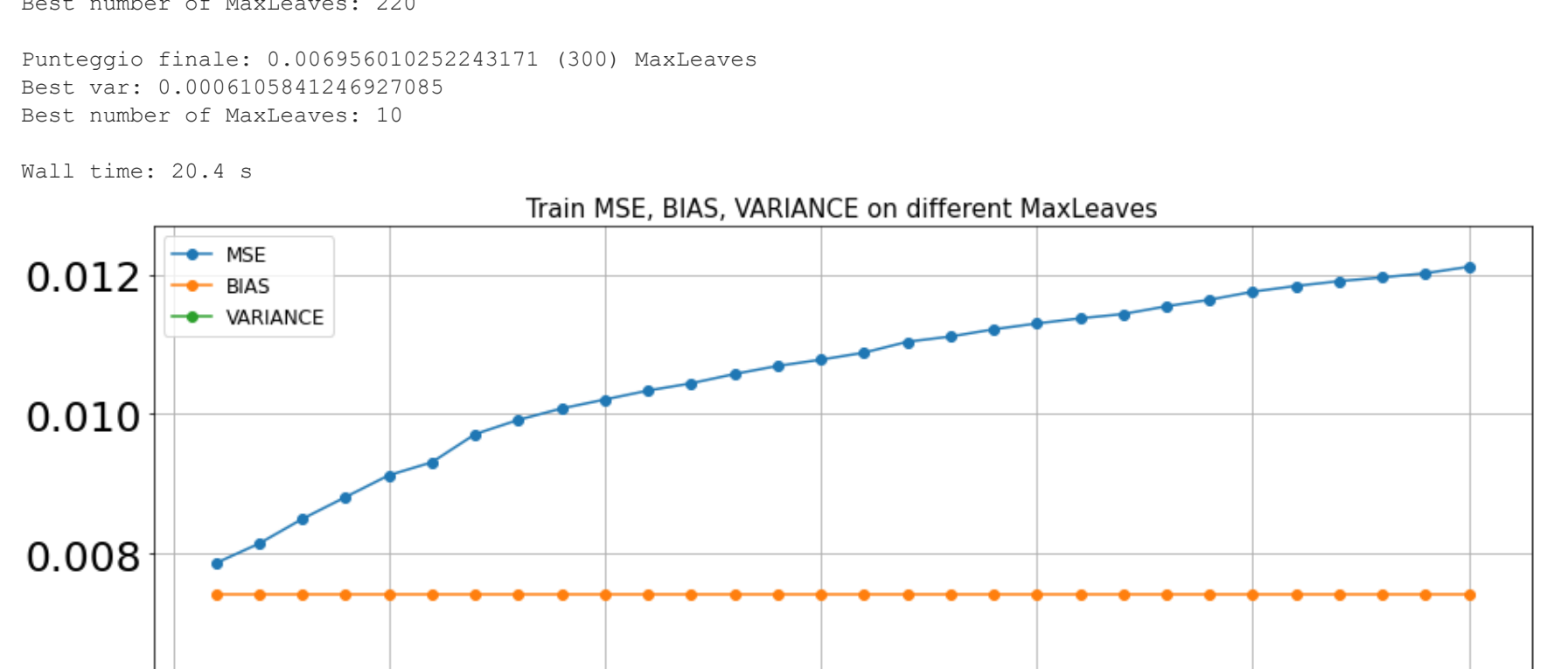


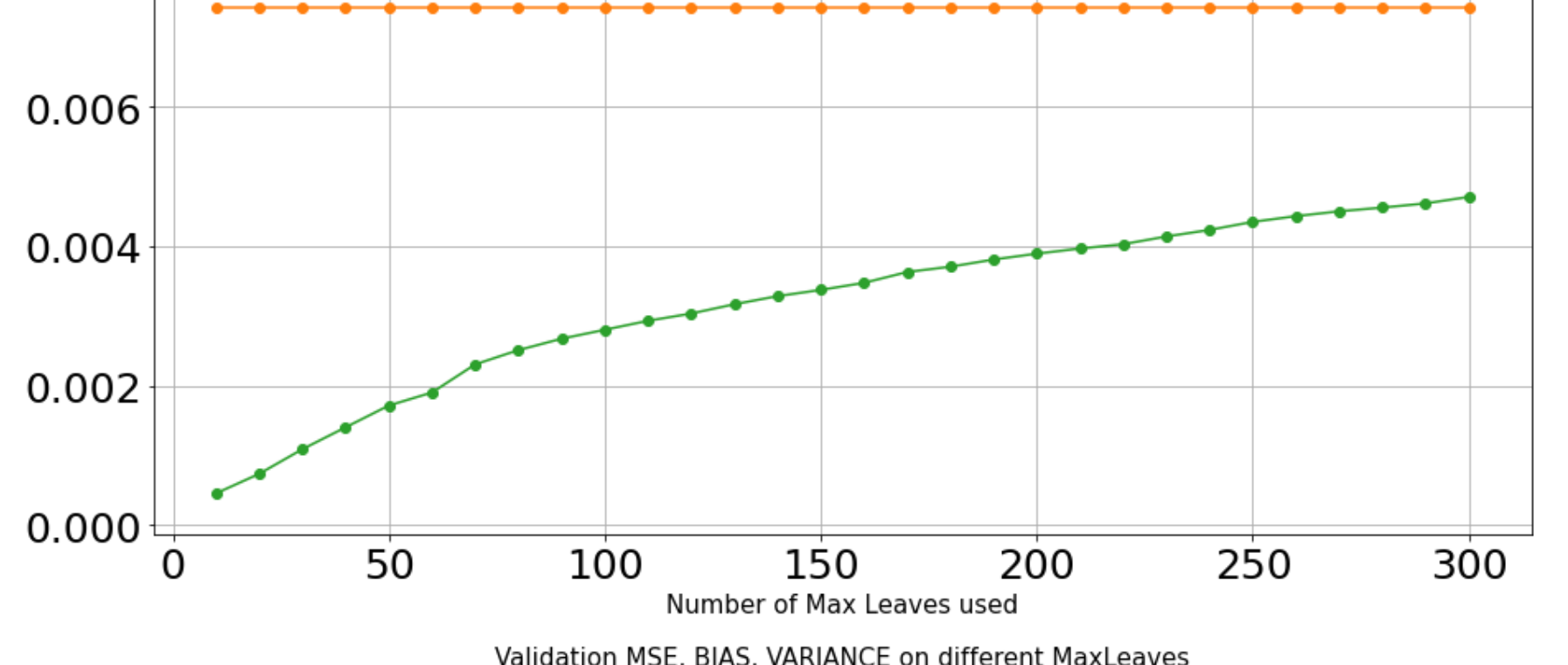


Seconda Regione 2061

In [20]:
```
%%time
dt_model.append(
    get_dt(
        1,
        verbose    = False,
        debug      = False#,
        #file_name = 'prova1'
    )
)
```

```
Train: TUNING DEL MASSIMO NUMERO DI FOGLIE

Punteggio finale: 0.012111755678008781 (300) MaxLeaves
Best mse: 0.007864440746398235
Best number of MaxLeaves: 10

Punteggio finale: 0.007408429688393943 (300) MaxLeaves
Best bias: 0.007408429688393943
Best number of MaxLeaves: 60

Punteggio finale: 0.004703325989614886 (300) MaxLeaves
Best var: 0.0004560110580042893
Best number of MaxLeaves: 10

Validation: TUNING DEL MASSIMO NUMERO DI FOGLIE

Punteggio finale: 0.022162968201747773 (300) MaxLeaves
Best mse: 0.015816528969226508
Best number of MaxLeaves: 10

Punteggio finale: 0.015206957949504602 (300) MaxLeaves
Best bias: 0.015205301853708214
Best number of MaxLeaves: 220

Punteggio finale: 0.006956010252243171 (300) MaxLeaves
Best var: 0.0006105841246927085
Best number of MaxLeaves: 10

Wall time: 20.4 s
```

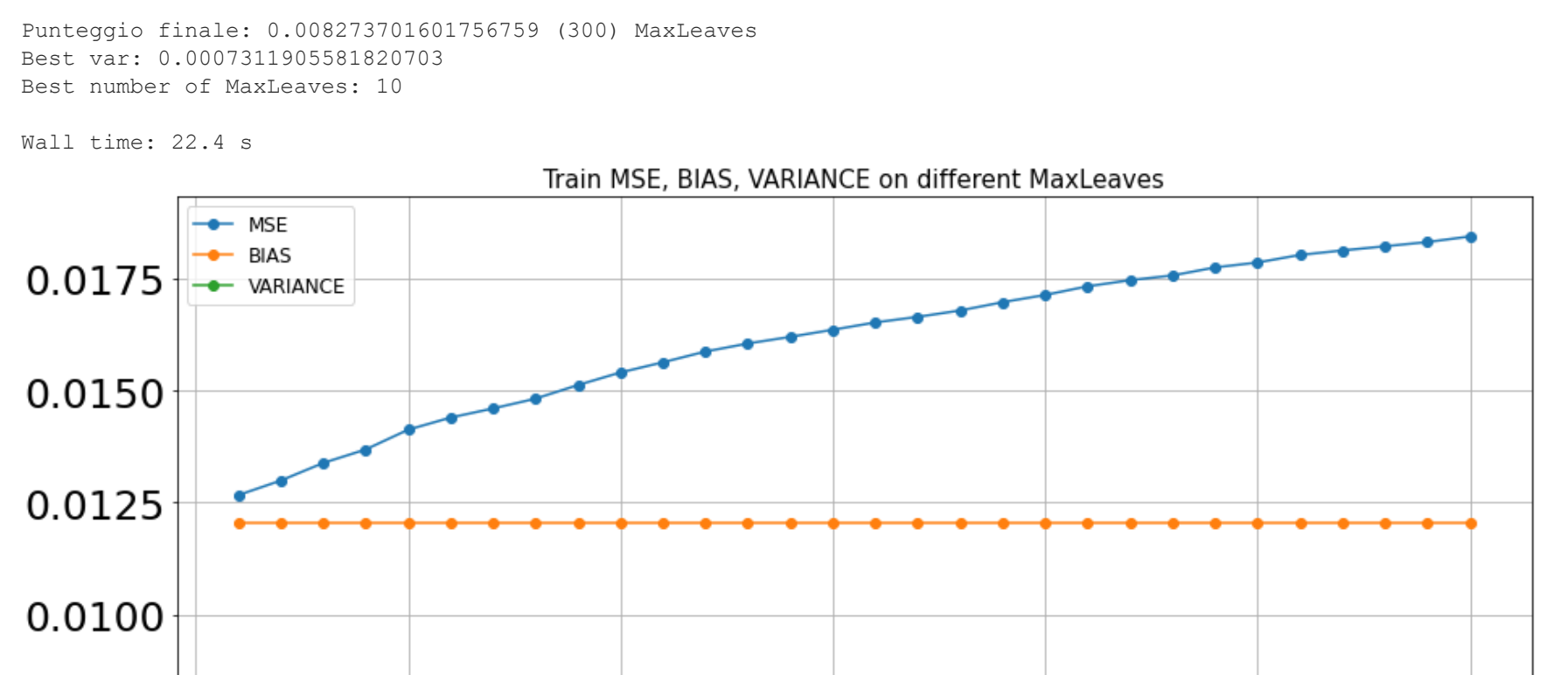


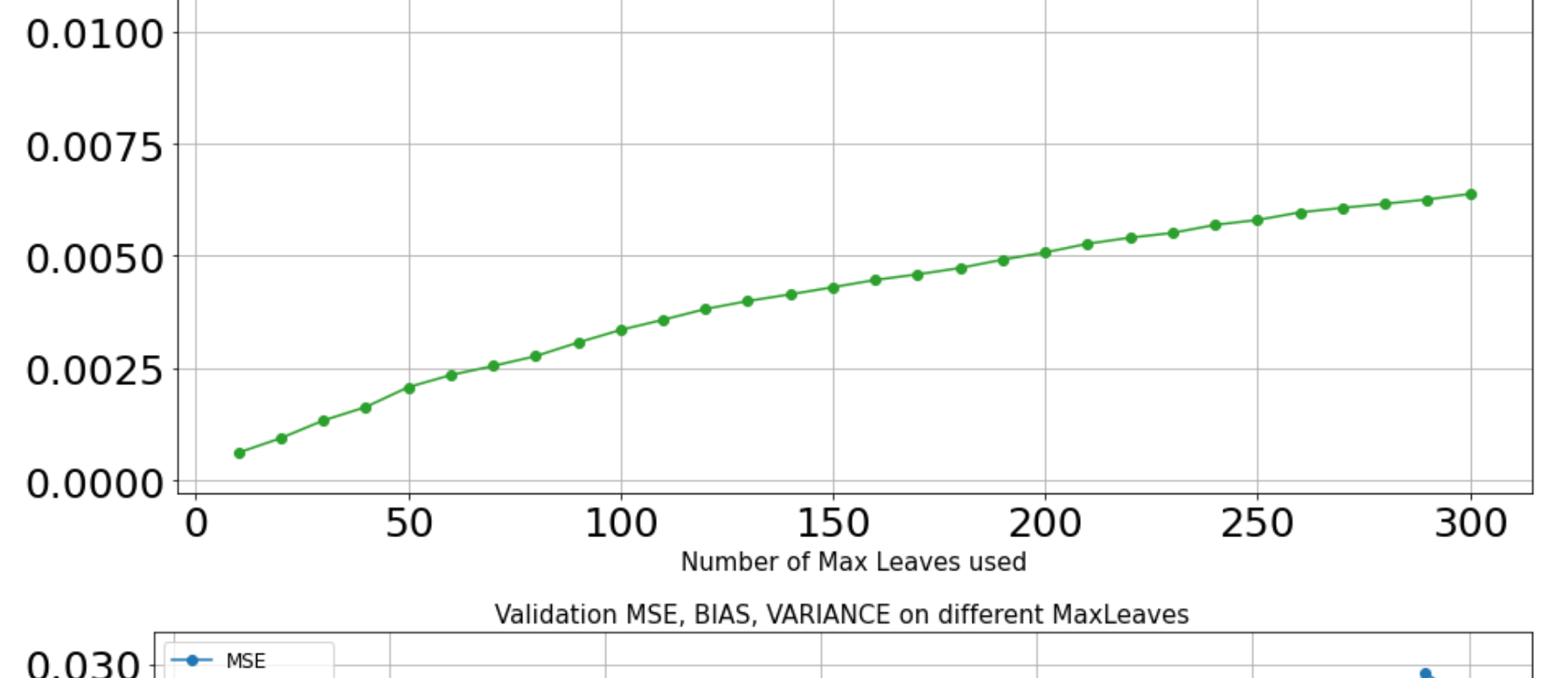


Terza Regione 3101

In [21]:
```
%%time
dt_model.append(
    get_dt(
        2,
        verbose    = False,
        debug      = False#,
        #file_name = 'prova2'
    )
)
```

```
Train: TUNING DEL MASSIMO NUMERO DI FOGLIE

Punteggio finale: 0.018429791191878 (300) MaxLeaves
Best mse: 0.012663186996844945
Best number of MaxLeaves: 10

Punteggio finale: 0.012049414522519323 (300) MaxLeaves
Best bias: 0.012049414522519323
Best number of MaxLeaves: 10

Punteggio finale: 0.0063803766693586355 (300) MaxLeaves
Best var: 0.0006137724743255895
Best number of MaxLeaves: 10

Validation: TUNING DEL MASSIMO NUMERO DI FOGLIE

Punteggio finale: 0.02857894831510075 (300) MaxLeaves
Best mse: 0.021017095137600226
Best number of MaxLeaves: 10

Punteggio finale: 0.02030524671334402 (300) MaxLeaves
Best bias: 0.020273881004273426
Best number of MaxLeaves: 280

Punteggio finale: 0.008273701601756759 (300) MaxLeaves
Best var: 0.0007311905581820703
Best number of MaxLeaves: 10

Wall time: 22.4 s
```





In [22]:
```
dt_model
```

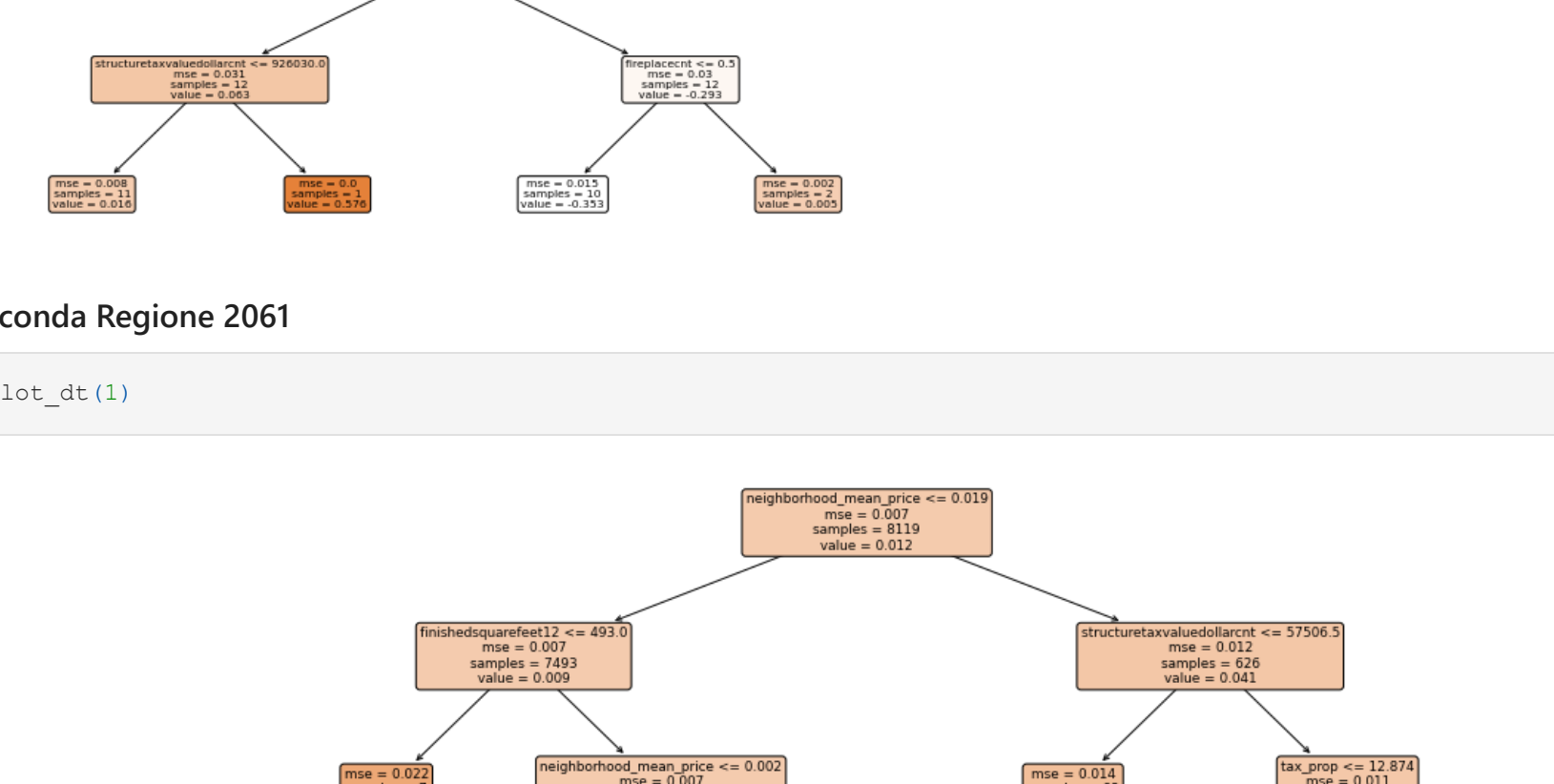
Out[22]:
```
[DecisionTreeRegressor(max_leaf_nodes=10),
 DecisionTreeRegressor(max_leaf_nodes=10),
 DecisionTreeRegressor(max_leaf_nodes=10)]
```

Visualizzazione degli alberi

In [23]:
```
def plot_dt(index, file_name=''):
    if file_name == '':
        file_name = region_ids[index]
    fig, ax = plt.subplots(figsize=(20,10))
    plot_tree(
        dt_model[index],
        ax = ax,
        feature_names= X_train[index].columns,
        filled=True,
        rounded=True,
    )
    if file_name != '':
        fig.savefig('images/' + file_name + '_DecionTreeRegressor.jpg')
```

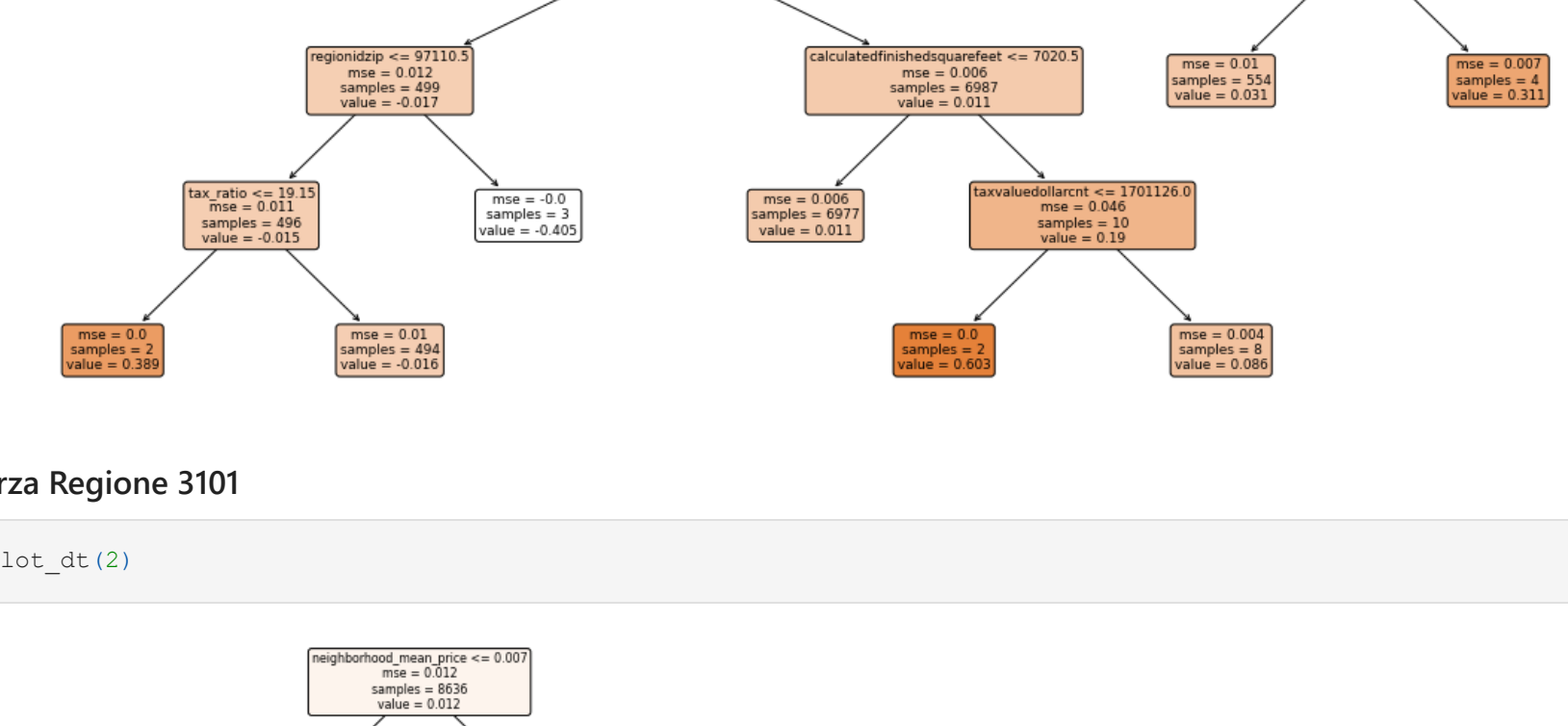
Prima Regione 1286

In [24]:
```
plot_dt(0)
```



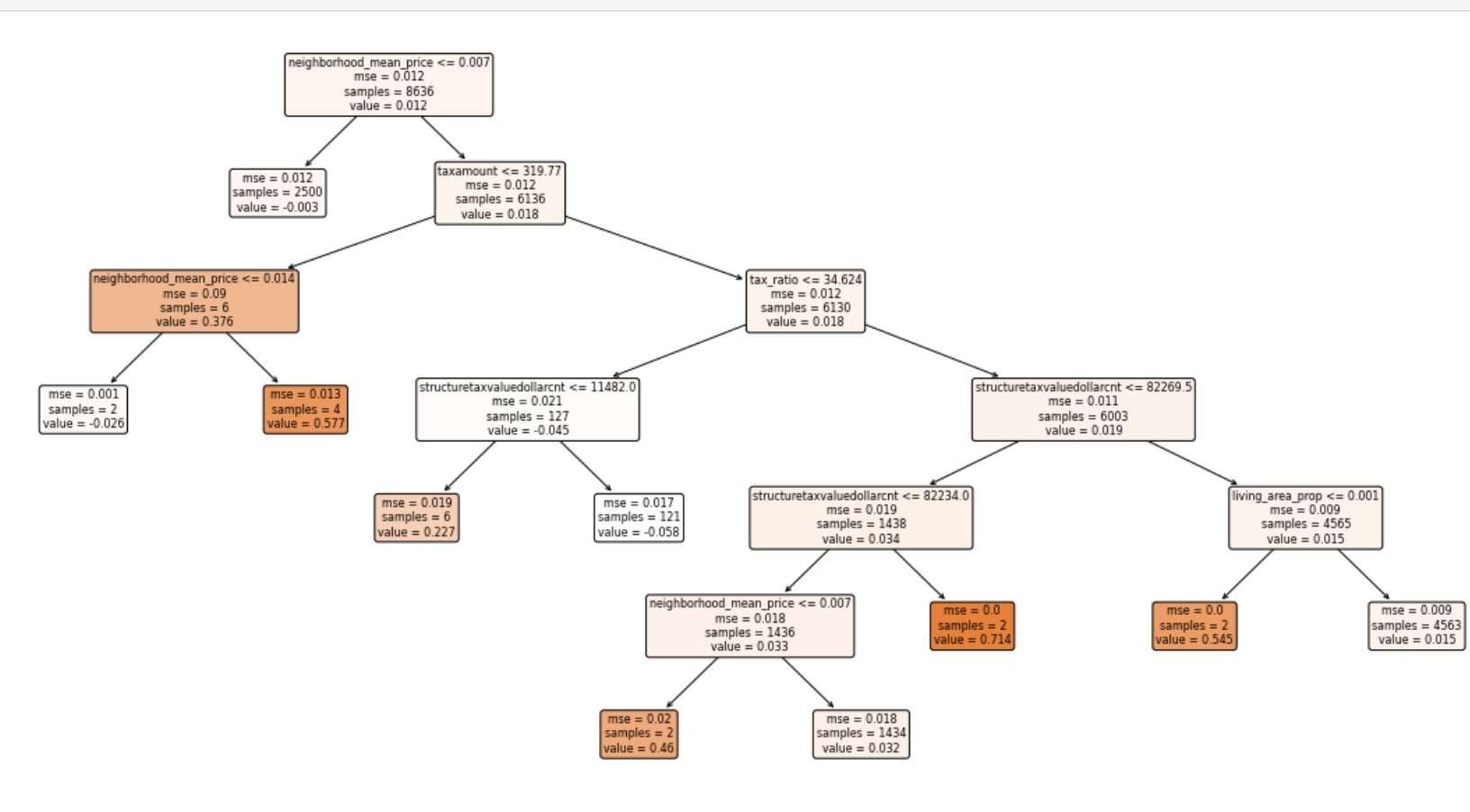
Seconda Regione 2061

In [25]:
```
plot_dt(1)
```



Terza Regione 3101

In [26]:
```
plot_dt(2)
```



Analisi dei risultati

I risultati ottenuti sono analoghi per tutte e tre le regioni; i risultati non sono per nulla soddisfacenti.

All'aumentare del numero di foglie, l'errore aumenta.

Decomponendo l'errore di può notare come questo errore provenga da una **varianza** che inevitabilmente tende a **crescere aumentando il numero di foglie**, e da un $bias^2$ che al posto che diminuire specializzandosi nei vari nodi **rimane costante**.

Il modello non riesce a imparare dai dati specializzandosi nei nodi, ma probabilmente fornisce indipendentemente dal nodo una predizione banale pari circa alla media delle osservazioni.

Analizzando l'albero si riesce a visualizzare come la struttura tenda a specializzarsi su una piccola fetta dei dati, lasciando un unica foglia per la stragrande maggioranza delle osservazioni per cui la predizione è molto simile alla media della totalità del dataset.

Il modello non riesce così ad abbattere la baseline del modello banale.

In [27]:
```python
def print_state(X, y, models):
    for i in range(3):
        print(f'{region_names[i]}: {get_bias_var_mse(X[i], y[i].values.ravel(), models[i])}')
        print()
```

In [28]:
```python
def print_state_trivial(X_train, y_train):
    y_pred = np.repeat(y_train.mean(), len(y_train))
    print( {
        'bias': ((y - np.mean(y_pred))**2).mean(),\
        'var':  np.var(y_pred).mean(),\
        'mse':  ((y_pred - y.reshape(-1,1))**2).mean()
    } )
```

In [29]:
```python
def print_all_stats(models):
    for X, y, name in zip(
        [X_train,         X_val,  X_test],
        [y_train,         y_val,  y_test],
        ['Train', 'Validation', 'Test']
    ):
        print(name)
        print()
        print_state(X, y, models)
        print()
```

In [30]:
```python
def print_all_state_trivial():
    for i in range(len(X_train)):
        print(region_names[i])
        print_state_trivial(y_train[i].values.ravel(), y_test[i].values.ravel())
```

Statistiche per le tre regioni su Train, Validation e Test

In [31]:
```python
print_all_stats(dt_model)
```

```
Train

A: {'bias': 0.0069294254023030055, 'var': 0.00042738828859140359, 'mse': 0.007356813688217095}

B: {'bias': 0.007536254902481718, 'var': 0.0003451420273853424, 'mse': 0.007881396915220221}

C: {'bias': 0.011674508124912, 'var': 0.0005817505209895822, 'mse': 0.01225625864148076}

Validation

A: {'bias': 0.02726635407649438, 'var': 0.000527347031855349, 'mse': 0.0277937011075504842}

B: {'bias': 0.019614365058197596, 'var': 0.00058068838112700375, 'mse': 0.020195044393245988}

C: {'bias': 0.02217266495119075, 'var': 0.0005495066299329632, 'mse': 0.022722171581123617}

Test

A: {'bias': 0.028641724215576377, 'var': 0.00030789391837899747, 'mse': 0.028949618133955471}

B: {'bias': 0.024158109782059628, 'var': 0.0002547132513325053, 'mse': 0.024412823033397434}

C: {'bias': 0.037762123652720005, 'var': 0.00047583974611381911, 'mse': 0.038237963398833999}
```

In [32]:
```python
for i in range(3):
    print(f'{region_names[i]}')
    print(pd.DataFrame(y_train[i]).describe())
    print()
```

```
A
          logerror
count  26819.000000
mean       0.012913
std        0.083244
min       -0.482335
25%       -0.017767
50%        0.006568
75%        0.034400
max        0.753300

B
          logerror
count   8119.000000
mean       0.013157
std        0.086804
min       -0.470000
25%       -0.020200
50%        0.006000
75%        0.036300
max        0.756800

C
          logerror
count  25908.000000
mean       0.012105
std        0.108051
min       -0.485174
25%       -0.029400
50%        0.005973
75%        0.041100
max        0.751627
```

L'errore quadratico medio è decisamente molto grande rispetto alla dimensionalità della risposta.

Statistiche per un modello banale sul test

In [33]:
```python
print_all_state_trivial()
```

```
A
{'bias': 0.028637396765634218, 'var': 1.2037062152420224e-35, 'mse': 0.028637396765634218}

B
{'bias': 0.024126424499493516, 'var': 3.0092655381050056e-36, 'mse': 0.024126424499493482}

C
{'bias': 0.037759230010362333, 'var': 0.0, 'var': 0.005773893150...
```

Gli errori dell'albero di decisione sono paragonabili a quelle del modello banale.

L'errore sembra provenire principalmente dal bias: si può pensare di sfruttare il **boosting** per ridurlo .

Boosting

Scelta l'algoritmo di boosting principalmente per due motivi:

- è un processo che tende a ridurre il $bias^2$ di un modello e l'errore degli alberi di decisione proviene principalmente dal bias
- è capace di raggiungere un buono score anche partendo da modelli relativamente semplici, come quelli selezionati dall'albero di decisione

Uso come modelli base gli alberi individuati ai punti precedenti

In [34]:
```python
base_model = []
#MAX_LEAF = 500
for dt in dt_model:
    base_model.append(DecisionTreeRegressor(max_leaf_nodes=dt.get_params()['max_leaf_nodes']))
    #base_model.append(DecisionTreeRegressor(max_leaf_nodes = MAX_LEAF))
base_model
```

Out[34]:
```
[DecisionTreeRegressor(max_leaf_nodes=10),
 DecisionTreeRegressor(max_leaf_nodes=10),
 DecisionTreeRegressor(max_leaf_nodes=10)]
```

In [35]:
```python
def boosting_train(X_train, y_train, X_val, y_val, baseModel, verbose=False, debug=False, file_name=''):

    def get_adaboost_regressor(Xs, ys, estimators):
        adaboost = AdaBoostRegressor(
            baseModel,
            n_estimators = estimators
        )
        adaboost.fit(Xs,ys)
        return adaboost

    def bias_var_mse(X, y, model):
        stats = get_bias_var_mse(X, y, model)
        return stats['bias'],\
               stats['var'],\
               stats['mse']

    def plot_mse(stats, name):

        print(f'{name}: TUNING DEL NUMERO DI STIMATORI')
        print()

        for n in ['mse', 'bias', 'var']:

            min_  = min(stats[n])
            best  = (np.argmin(stats[n]) * BOOST_STEP) + BOOST_START

            print(f'Punteggio finale: {stats[n][-1]} ({BOOST_END}) stimatori')
            print(f'Best {n}:, {min_}')
            print(f'Best number of Estimators: {best}')
            print()

        fig, ax = plt.subplots(figsize=(len(stats['mse'])/2, 10))

        ax.tick_params(axis='both', which='major', labelsize=25)
        ax.tick_params(axis='both', which='minor', labelsize=15)

        ax.plot(range(BOOST_START, BOOST_END+1, BOOST_STEP),  stats['mse'], 'o-', label='MSE')
        ax.plot(range(BOOST_START, BOOST_END+1, BOOST_STEP),  stats['bias'], 'o-', label='BIAS')
        ax.plot(range(BOOST_START, BOOST_END+1, BOOST_STEP),  stats['var'], 'o-', label='VARIANCE')

        ax.set_title(f'{name}: MSE, BIAS, VARIANCE on different Estimators', fontsize=15)
        ax.set_xlabel("Number of Max Estimators used", fontsize=15)
        ax.grid()
        ax.legend(prop={'size': 12})

        if file_name != '':
            fig.savefig('images/' + file_name + '_AdaBoostRegressor_' + name + '.jpg')


    stats   = np.array([])
    boosts  = range(BOOST_START, BOOST_END+1, BOOST_STEP)

    y_train = y_train.values.ravel()
    y_val   = y_val  .values.ravel()

    first = True

    info = []

    train_stats = {
        'bias' : [],
        'var'  : [],
        'mse'  : []
    }

    val_stats = {
        'bias' : [],
        'var'  : [],
        'mse'  : []
    }

    for b in boosts:

        if debug:
            print(f'{b}/{BOOST_END}')

        train_stats_s = {
            'bias' : [],\
            'var'  : [],\
            'mse'  : []
        }

        val_stats_s = {
            'bias' : [],\
            'var'  : [],\
            'mse'  : []
        }

        # Resampling
        for i in range(BOOST_NTEST):

            if debug:
                print(f'      {i+1}/{BOOST_NTEST}')

            X_sample, y_sample = resample(X_train, y_train, n_samples = int(BOOST_SAMPLE_PERC*len(y_train)))

            ada = get_adaboost_regressor(X_sample, y_sample, b)

            trn_bias, trn_var, trn_mse = bias_var_mse(X_train, y_train, ada)
            val_bias, val_var, val_mse = bias_var_mse(  X_val,   y_val, ada)

            train_stats_s['bias'].append(trn_bias)
            train_stats_s['var' ].append(trn_var )
            train_stats_s['mse' ].append(trn_mse )

            val_stats_s  ['bias'].append(val_bias)
            val_stats_s  ['var' ].append(val_var )
            val_stats_s  ['mse' ].append(val_mse )


        trn_bias_s = np.array(train_stats_s['bias']).mean()
        trn_var_s  = np.array(train_stats_s['var' ]).mean()
        trn_mse_s  = np.array(train_stats_s['mse' ]).mean()

        val_bias_s = np.array(  val_stats_s['bias']).mean()
        val_var_s  = np.array(  val_stats_s['var' ]).mean()
        val_mse_s  = np.array(  val_stats_s['mse' ]).mean()


        train_stats['bias'].append(trn_bias_s)
        train_stats['var' ].append(trn_var_s )
        train_stats['mse' ].append(trn_mse_s )

        val_stats   ['bias'].append(val_bias_s)
        val_stats   ['var' ].append(val_var_s )
        val_stats   ['mse' ].append(val_mse_s )


        info.append (
            f"\n{Train MSE:      {trn_mse_s } - Val MSE:      {val_mse_s }}" +\
            f"\n{Train Bias:     {trn_bias_s} - Val Bias:     {val_bias_s}}" +\
            f"\n{Train Variance: {trn_var_s } - Val Variance: {val_var_s }}"
        )


        if(first or val_mse_s < best_mse):
            first = False
            best_mse = val_mse_s
            best_estimators = b

    if verbose:
        print(*info, sep='\n')

    plot_mse(train_stats, 'Train')
    plot_mse(val_stats, 'Validation')

    return get_adaboost_regressor(X_train, y_train, best_estimators)
```

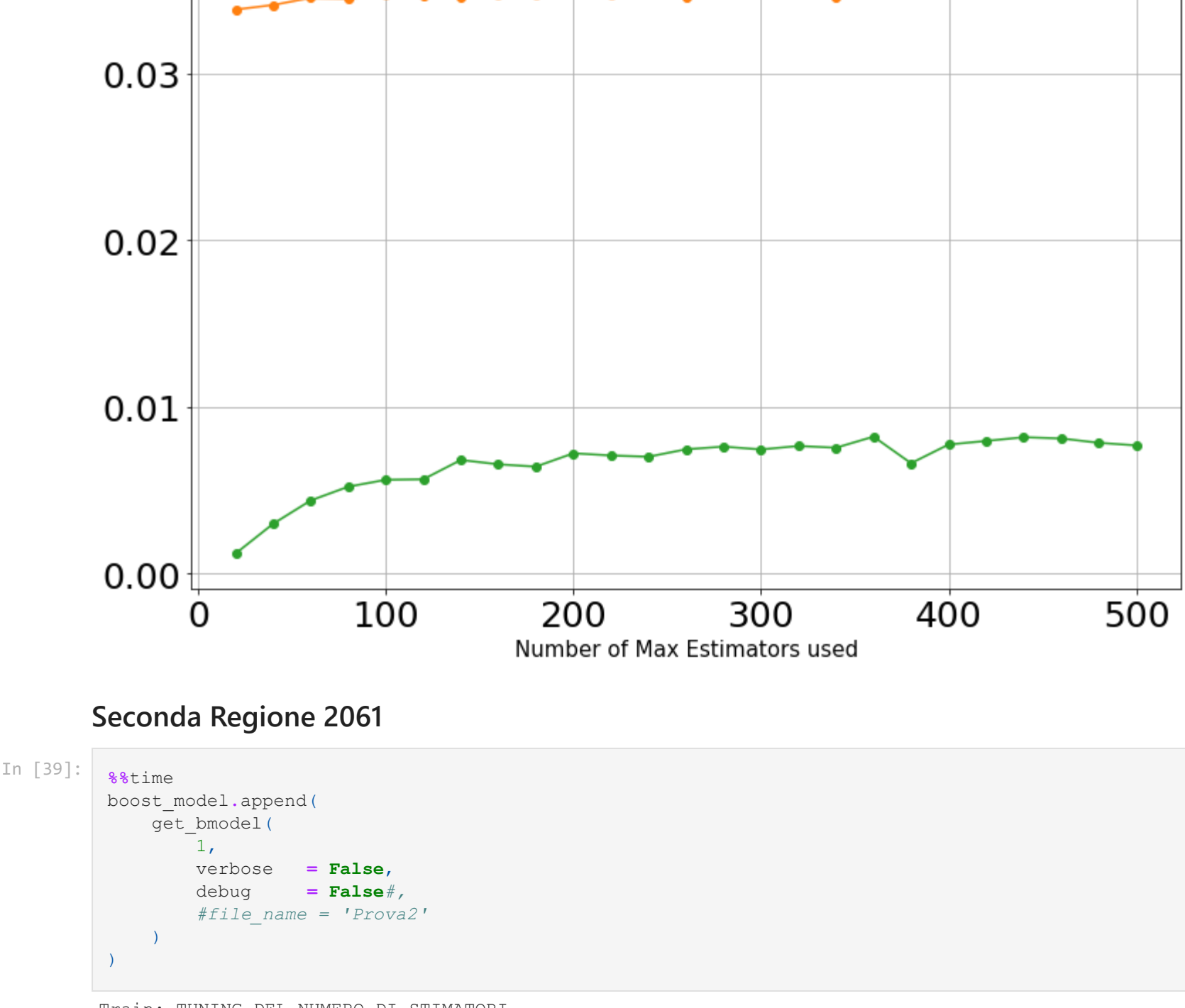
In [36]:
```python
boost_model = []
```

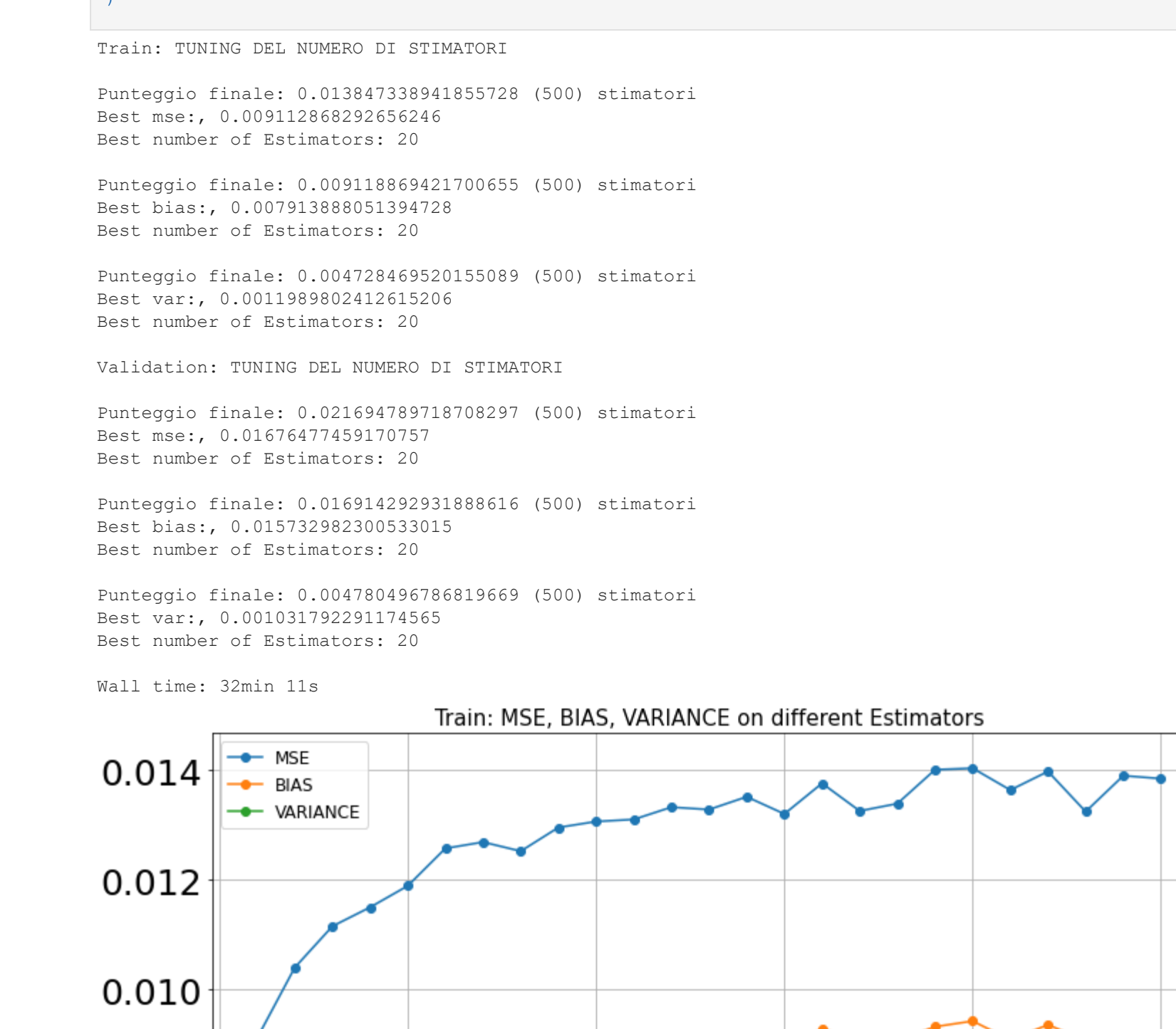
In [37]:
```python
def get_bmodel(index, verbose=False, debug=False, file_name=''):
    if file_name == '':
        file_name = region_ids[index]
    return boosting_train(
        X_train_sub [index],
        y_train_sub [index],
        X_val_sub   [index],
        y_val_sub   [index],
        base_model  [index],
        verbose    = verbose,
        debug      = debug,
        file_name = file_name
    )
```

Prima Regione 1286

In [38]:
```python
%%time
boost_model.append(
    get_bmodel(
        0,
        verbose    = False,
        debug      = False#,
        #file_name = 'Prova1'
    )
)
```

```
Train: TUNING DEL NUMERO DI STIMATORI

Punteggio finale: 0.016299857197165746 (500) stimatori
Best mse:, 0.008570762096412658
Best number of Estimators: 20

Punteggio finale: 0.008661629437220689 (500) stimatori
Best bias:, 0.007319919027574296
Best number of Estimators: 20

Punteggio finale: 0.007638227759945063 (500) stimatori
Best var:, 0.001250843068838356
Best number of Estimators: 20

Validation: TUNING DEL NUMERO DI STIMATORI

Punteggio finale: 0.04272946164679128 (500) stimatori
Best mse:, 0.03505686725975199
Best number of Estimators: 20

Punteggio finale: 0.03504045059819914 (500) stimatori
Best bias:, 0.03385121053172969
Best number of Estimators: 20

Punteggio finale: 0.007689011048592151 (500) stimatori
Best var:, 0.0012056567280222977
Best number of Estimators: 20

Wall time: 45min 32s
```

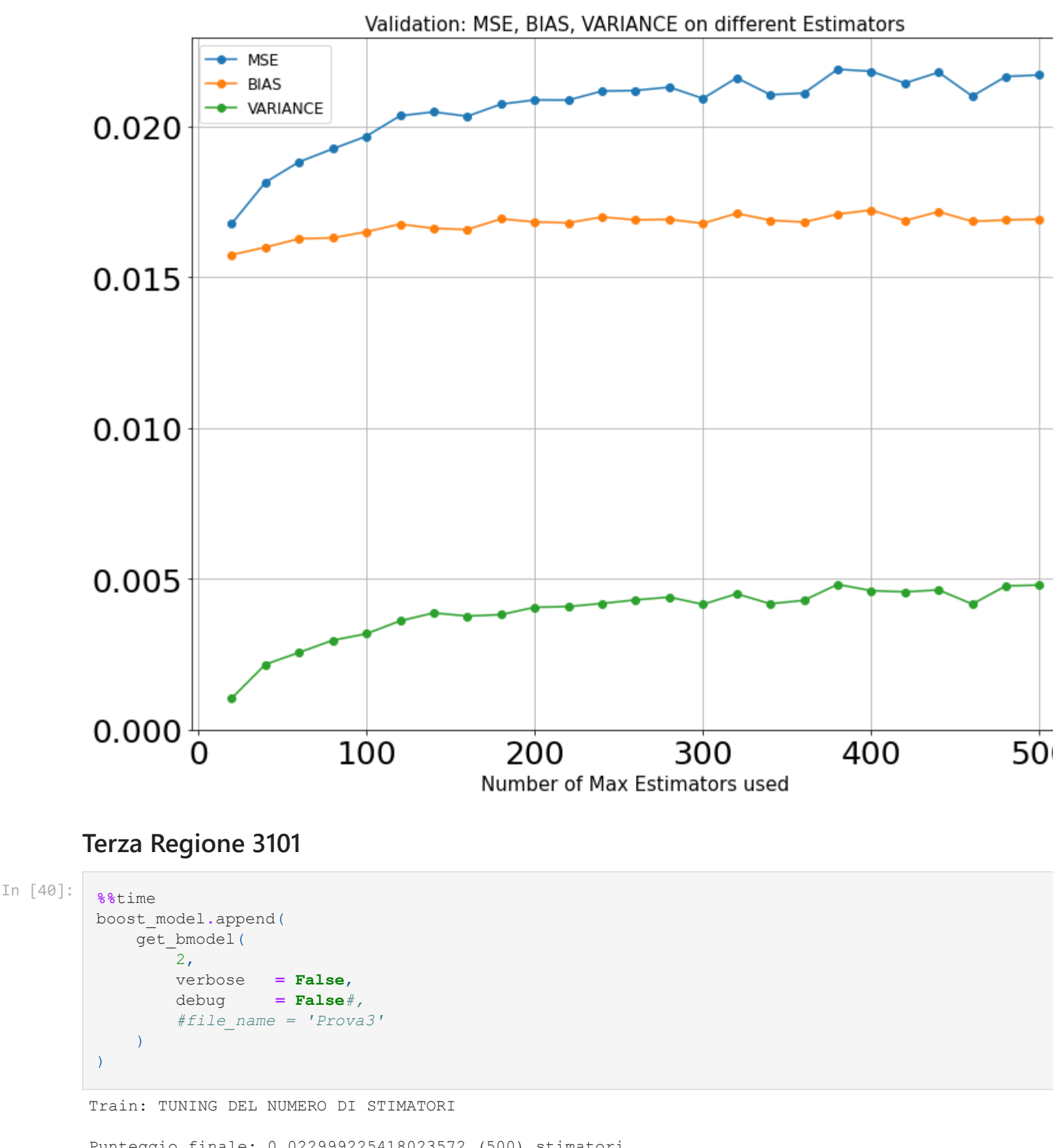




Seconda Regione 2061

In [39]:
```python
%%time
boost_model.append(
    get_bmodel(
        1,
        verbose    = False,
        debug      = False#,
        #file_name = 'Prova2'
    )
)
```

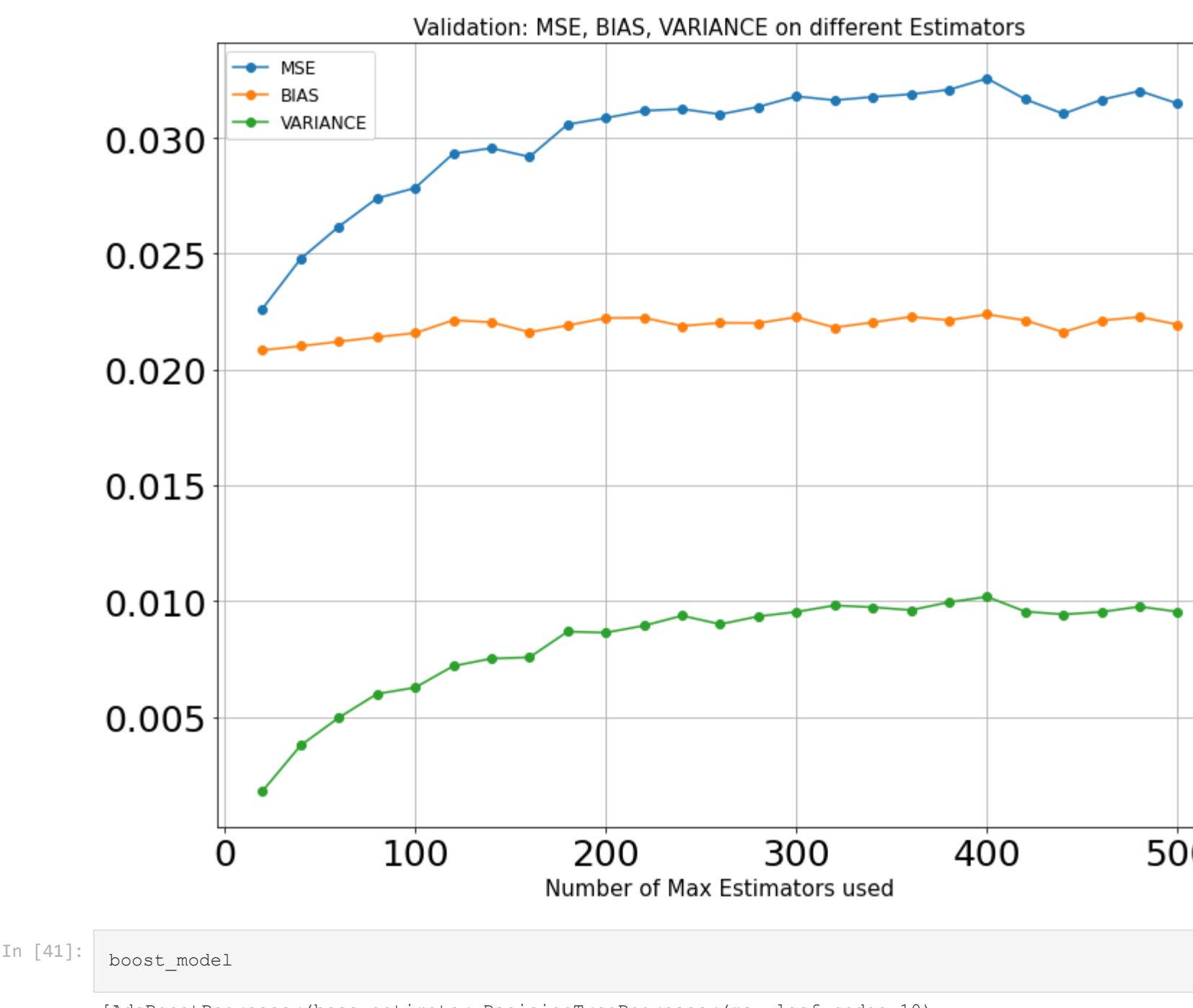
```
Train: TUNING DEL NUMERO DI STIMATORI

Punteggio finale: 0.013847338941855728 (500) stimatori
Best mse:, 0.009112868292656246
Best number of Estimators: 20

Punteggio finale: 0.009118869421700655 (500) stimatori
Best bias:, 0.007913888051394728
Best number of Estimators: 20

Punteggio finale: 0.004728469520155089 (500) stimatori
Best var:, 0.001198980241261520 6
Best number of Estimators: 20

Validation: TUNING DEL NUMERO DI STIMATORI

Punteggio finale: 0.021694789718708297 (500) stimatori
Best mse:, 0.01676477459170757
Best number of Estimators: 20

Punteggio finale: 0.016914292931888616 (500) stimatori
Best bias:, 0.015732982300533015
Best number of Estimators: 20

Punteggio finale: 0.004780496786819669 (500) stimatori
Best var:, 0.001031792291174565
Best number of Estimators: 20

Wall time: 32min 11s
```



Terza Regione 3101

In [40]:
```python
%%time
boost_model.append(
    get_bmodel(
        2,
        verbose    = False,
        debug      = False#,
        #file_name = 'Prova3'
    )
)
```

```
Train: TUNING DEL NUMERO DI STIMATORI

Punteggio finale: 0.022999225418023572 (500) stimatori
Best mse:, 0.014459069422703771
Best number of Estimators: 20

Punteggio finale: 0.013376148868989993 (500) stimatori
Best bias:, 0.012476381051324944
Best number of Estimators: 20

Punteggio finale: 0.009623076728123642 (500) stimatori
Best var:, 0.001982688371378823 8
Best number of Estimators: 20

Validation: TUNING DEL NUMERO DI STIMATORI

Punteggio finale: 0.031473346645053434 (500) stimatori
Best mse:, 0.022620905013511113
Best number of Estimators: 20

Punteggio finale: 0.021936199918809712 (500) stimatori
Best bias:, 0.02082657073749757
Best number of Estimators: 20

Punteggio finale: 0.009537146531724628 (500) stimatori
Best var:, 0.001794334276013548
Best number of Estimators: 20

Wall time: 36min 55s
```



In [41]:
```python
boost_model
```

Out[41]:
```
[AdaBoostRegressor(base_estimator=DecisionTreeRegressor(max_leaf_nodes=10),
                   n_estimators=20),
 AdaBoostRegressor(base_estimator=DecisionTreeRegressor(max_leaf_nodes=10),
                   n_estimators=20),
 AdaBoostRegressor(base_estimator=DecisionTreeRegressor(max_leaf_nodes=10),
                   n_estimators=20)]
```

Analisi dei risultati

I risultati si distaccano da quelli sperati: non si evidenzia una significativa diminuzione del bias, che sembra non riuscire ad abbattere una certa soglia limite. Il problema principale è analogo quello precedente: l'albero non sembra riuscire a imparare dai dati e a specializzarsi in maniera utile.

Statistiche per le tre regioni su Train, Validation e Test

In [42]:
```python
print_all_stats(boost_model)
```

```
Train

A: {'bias': 0.007573332607368333, 'var': 0.0012659292355888122, 'mse': 0.008839261843256509}

B: {'bias': 0.007815710232407217, 'var': 0.0009322285297467683, 'mse': 0.008747938762153967}

C: {'bias': 0.012297459946746645, 'var': 0.0014159475130206248, 'mse': 0.013713407453497 29}

Validation

A: {'bias': 0.027867041028237897, 'var': 0.0012991623840724967, 'mse': 0.029166203412310 31}

B: {'bias': 0.019782280858029664, 'var': 0.0007790469655768118, 'mse': 0.020561327823606486}

C: {'bias': 0.022993356150102243, 'var': 0.0014057924097579516, 'mse': 0.024399148559860275}

Test

A: {'bias': 0.029003795884508705, 'var': 0.0012252205334630516, 'mse': 0.030290164179669298}

B: {'bias': 0.024145364484936275, 'var': 0.0007621476849189092, 'mse': 0.024907512179281 66}

C: {'bias': 0.038217943882926 29, 'var': 0.0013890627394409 26, 'mse': 0.039607006222404005}
```

Come atteso dalle precedenti analisi, non sembra che il boosting sia riuscito a diminuire l'errore, questo perché non è riuscito a diminuire come sperato in maniera significativa il bias, che sembra non riuscire ad abbattere anche con il boosting una certa soglia limite pari a quella del bias del predittore banale.