

January 8, 2026

1 A quick, rough example of the convenience of random forest

Warning: This is *not* how we actually do real statistical predictive modeling.

```
[1]: from pathlib import Path
import tarfile
import urllib.request

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
```

1.1 1. Get data

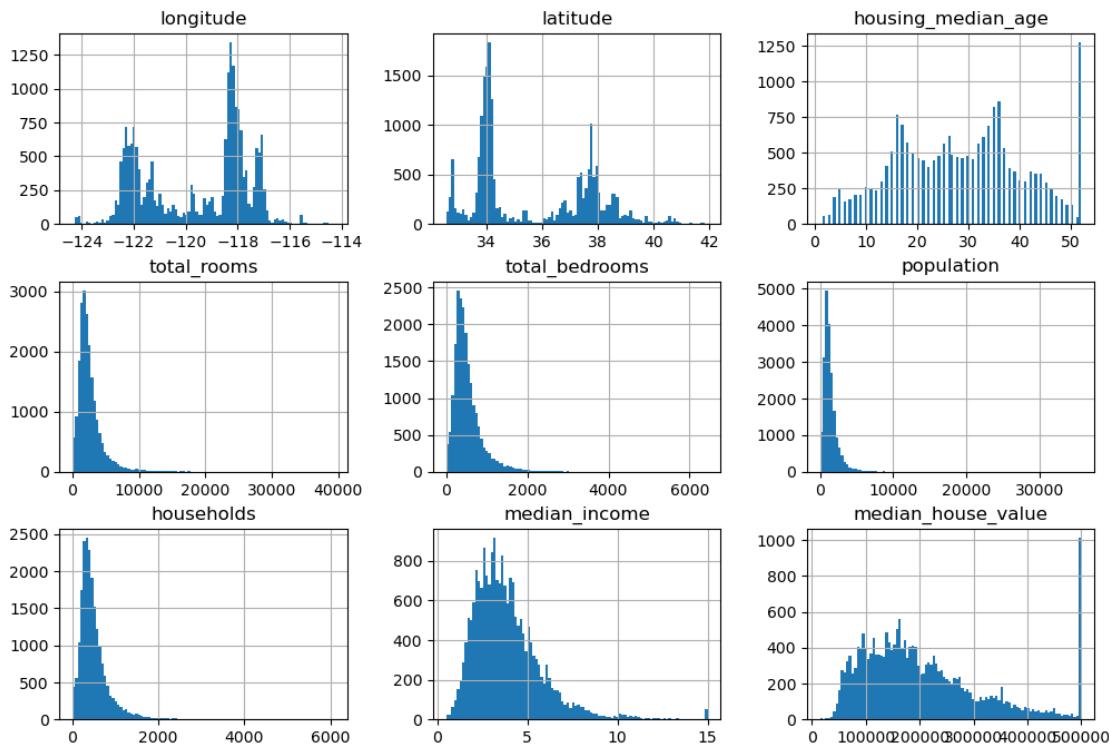
```
[2]: def get_data():
    path_tar = Path('data') / 'housing.tgz'
    if not path_tar.is_file():
        Path('data').mkdir(parents=True, exist_ok=True)
        url = 'https://github.com/ageron/data/raw/main/housing.tgz'
        urllib.request.urlretrieve(url, path_tar)
        with tarfile.open(path_tar) as housing_tar:
            housing_tar.extractall(path='data')
    return pd.read_csv(Path('data') / 'housing' / 'housing.csv')

df = get_data()
```

```
[3]: print(f'housing data shape: {df.shape}\n')
print(df.info())
df.hist(bins=100, figsize=(12, 8))
plt.show()
```

```
housing data shape: (20640, 10)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   longitude        20640 non-null   float64
 1   latitude         20640 non-null   float64
 2   housing_median_age 20640 non-null   float64
 3   total_rooms       20640 non-null   float64
 4   total_bedrooms    20433 non-null   float64
 5   population        20640 non-null   float64
 6   households        20640 non-null   float64
 7   median_income     20640 non-null   float64
 8   median_house_value 20640 non-null   float64
 9   ocean_proximity   20640 non-null   object  
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
None
```



In real work, there would be a lot to do here. For example, the variable total_bedrooms has missing values; many of these variables have skewed distribution, so should we transform them (perhaps with logs?); a few are also clearly censored at the top; and what units are they in, anyway? On

top of that, there is a categorical variable, ocean proximity, that should be encoded appropriately.

Here, we will ignore all of these issues: we will drop observations with missing values and work only with numeric variables. We will also ignore data censoring and a series of other issues as the purpose of this notebook is to give a very general notion of the convenience of random forests.

```
[4]: df_num = df.select_dtypes(include=np.number).dropna()
std_scaler = StandardScaler()
df_num_scaled = std_scaler.fit_transform(df_num)
train_df, test_df = train_test_split(df_num_scaled, test_size=0.2, random_state=2026)
train_df = pd.DataFrame(data=train_df, columns=df_num.columns)
test_df = pd.DataFrame(data=test_df, columns=df_num.columns)
y_train = train_df['median_house_value']
X_train = train_df.drop('median_house_value', axis=1)
y_test = test_df['median_house_value']
X_test = test_df.drop('median_house_value', axis=1)
```

1.2 2. Linear regression

We know that the unadjusted R^2 of a linear regression improves with the number of regressors; so we will throw all regressors at this model. Before we judge linear regression too harshly, though, remember there is a lot more we could do with feature engineering and, since we're trying to predict, with estimate shrinkage.

```
[5]: model_lr = LinearRegression(
    fit_intercept=False
)
model_lr.fit(X_train, y_train)

print('Linear regression')
print(f'In-sample R^2: {model_lr.score(X_train, y_train)}')
print(f'Out-of-sample R^2: {model_lr.score(X_test, y_test)}')
```

```
Linear regression
In-sample R^2: 0.6379930078949403
Out-of-sample R^2: 0.6308976771953473
```

Perhaps not bad given how little we did to help the model. As an interesting sidenote, the out-of-sample prediction is almost as good as the in-sample. Altering a little the train / test split can even make the out-of-sample score higher than the in-sample! This probably means that explanatory variables are doing almost no work and most of the predictions are around the mean value (of zero, here), but I haven't bothered to check.

2 3. Random forest

2.1 3.1 A quick, off-the-cuff random forest

```
[6]: model_rndfor = RandomForestRegressor(  
        n_estimators=100,  
        criterion='squared_error',  
        max_depth=None,  
        min_samples_leaf=2,  
        random_state=2026,  
        n_jobs=-1  
)  
model_rndfor.fit(X_train, y_train)  
  
print('Random forest regression')  
print(f'In-sample R^2: {model_rndfor.score(X_train, y_train)}')  
print(f'Out-of-sample R^2: {model_rndfor.score(X_test, y_test)}')
```

```
Random forest regression  
In-sample R^2: 0.9582667793399272  
Out-of-sample R^2: 0.8165208878893362
```

This one already perform a lot better than linear regression. Our only cost was a few seconds of time (and the compute cost).

3 3.2 A first pass at hyperparameter tuning

Note that there's quite a few things we would do before parameter tuning, but the idea here is to go back to thinking about what makes ensemble models good.

```
[7]: model_rndfor_template = RandomForestRegressor(  
        max_depth=None,  
        min_samples_leaf=2,  
        criterion='squared_error',  
        random_state=2026,  
        n_jobs=-1  
)  
  
param_grid = [{  
    'n_estimators': range(1, 201),  
}]  
  
grid_search = GridSearchCV(model_rndfor_template, param_grid, cv=5)  
grid_search.fit(X_train, y_train)  
print(grid_search.best_params_)  
  
{'n_estimators': 196}
```

```
[8]: model_rndfor_opt = RandomForestRegressor(
    max_depth=None,
    min_samples_leaf=2,
    n_estimators=196,
    criterion='squared_error',
    random_state=2026,
    n_jobs=-1
)
model_rndfor_opt.fit(X_train, y_train)
print('Random forest regression, optimized by number of tree')
print(f'In-sample R^2: {model_rndfor_opt.score(X_train, y_train)}')
print(f'Out-of-sample R^2: {model_rndfor_opt.score(X_test, y_test)}')
```

Random forest regression, optimized by number of tree

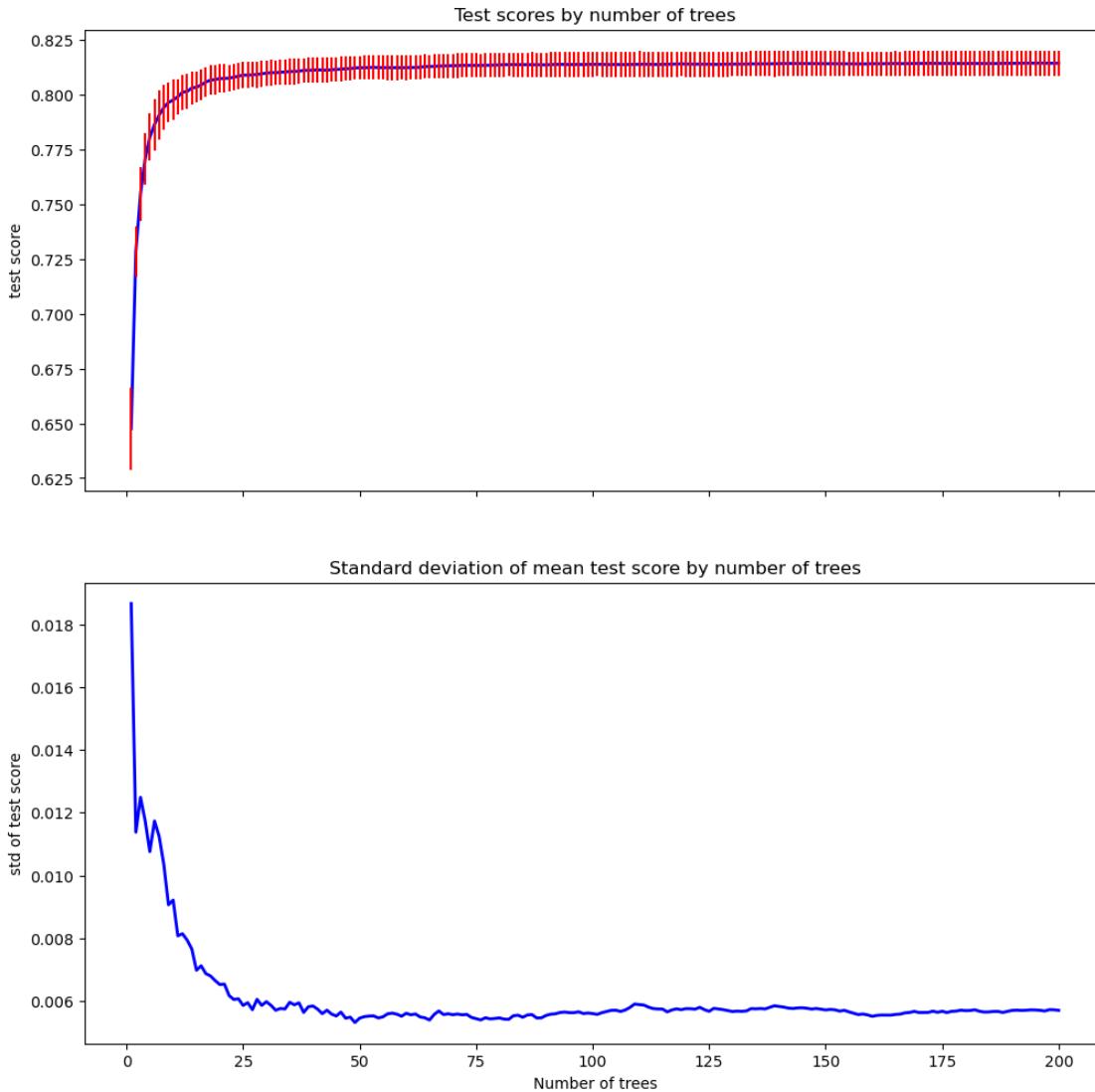
In-sample R²: 0.9585287614042396

Out-of-sample R²: 0.8173745451141257

Anecdotally, we lucked out with our first guess at the model: though the best results are happening with 196 trees, the improvement over our first “guess” at 100 trees is just as good for practical purposes!

I tried grid searches with some other hyper parameters, and we seemed to have just about nailed it by sheer luck with our toy example.

```
[9]: fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True, figsize=(12, 12))
grid_df = pd.DataFrame(data=grid_search.cv_results_)
grid_df = grid_df.set_index(grid_df['param_n_estimators'])
# Create the plot
error_bar = [grid_df['std_test_score'], grid_df['std_test_score']]
ax0.plot(grid_df['param_n_estimators'], grid_df['mean_test_score'], □
         linestyle='-', color='blue', linewidth=2)
ax0.errorbar(grid_df['param_n_estimators'], grid_df['mean_test_score'], □
             yerr=error_bar, fmt='none', color='r')
ax0.set_title('Test scores by number of trees')
ax0.set_ylabel('test score')
ax1.set_ylabel('std of test score')
ax1.plot(grid_df['param_n_estimators'], grid_df['std_test_score'], □
         linestyle='-', color='blue', linewidth=2)
ax1.set_title('Standard deviation of mean test score by number of trees')
plt.xlabel('Number of trees')
plt.show()
#grid_df.plot.scatter(y='split0_test_score', x='param_n_estimators')
```



3.1 4. So much left to do!

- A much smaller random forest seems to be “good” enough, maybe around 50 trees?
- How about tree depth or number of obs at leaves? How about number of regressors to pick from at each split?
- And how about boosting or other such improvements?
- Nevermind some feature engineering, including PCA, imputation of missing, etc?

So stay tuned for next year!

[]: