# Practice: From Single Decision Trees to Random Forests and Boosting

## Overview

You will begin by working with a real-world dataset from the 2016 US Presidential Election at the county level, learning how to predict the voting gap between candidates based on demographic features. After training simple Decision Tree models (using `DecisionTreeRegressor`), you will move on to ensemble methods such as Bagging (using multiple `DecisionTreeRegressor` models combined) and then to Random Forests (using `RandomForestRegressor`). Finally, you will experiment with Boosting techniques (using `AdaBoostRegressor`).

Before we begin, recall:

- **Decision Trees:** A flexible, interpretable model but prone to overfitting if left unchecked.
- **Bagging:** An approach to reduce variance by averaging predictions from multiple models trained on different bootstrap samples.
- **Random Forests:** A powerful ensemble method that introduces randomness both in data and features, typically outperforming simple bagging by reducing correlation between individual trees.
- **Boosting:** A method that builds an ensemble model in a sequential manner, where each new model focuses on the errors of the previous ones, often resulting in highly accurate predictions.

---

## Section 1: Exploring the Dataset

*Introduction:*
Before modeling, we need to understand our data. Good predictions often start with a thorough understanding of predictors and the response variable. This step helps clarify the modeling strategy and consider potential transformations.

**Task:** Load the dataset and explore the features and response.

- Use `pd.read_csv` to load the dataset.
- Use `train_test_split` from `sklearn.model_selection` to create training and test sets.

- Use `DataFrame.hist()` from `pandas` and `matplotlib` to visualize and understand the distribution of demographic features and the response (`votergap`).

**Reading and splitting data** is the first step in any ML workflow. By ensuring we have distinct training and testing sets, we can fairly assess how well our model generalizes to unseen data.

**Main functions/methods:**

- `pd.read_csv`
- `train_test_split`
- `DataFrame.hist()`

---

## Section 2: Building a Single Decision Tree

*Introduction:*
**Decision Trees** are an intuitive starting point. They recursively split the data to minimize predictive error. However, a single tree might overfit and not generalize well. This simple model will set a baseline for comparison with more advanced methods.

**Task:**

1. Fit a `DecisionTreeRegressor` on the training data (`Xtrain, ytrain`).
2. Evaluate its performance on both training and test sets using `score()` to understand overfitting.
3. Perform a brief parameter tuning experiment using cross-validation (`cross_val_score`) to find a good `max_depth` for the tree.

**Cross-validation (CV)** allows us to gauge how well a model generalizes beyond the training data. By averaging performance over multiple folds, we gain insight into the model's stability and reduce the risk of overfitting to a single train-test split.

**Main functions/methods:**

- `DecisionTreeRegressor()`
- `cross_val_score()`
- `fit()`, `predict()`, `score()` methods of the regressor.

---

## Section 3: Bagging: Bootstrap Aggregation

*Introduction:*
**Bagging (Bootstrap Aggregation)** builds many independent models on different bootstrap samples of the data and averages their predictions. This approach reduces variance and stabilizes predictions. While each individual tree might be noisy, their average is often smoother and more robust.

**Task:**

1. Using `resample()` from `sklearn.utils`, repeatedly draw bootstrap samples from `Xtrain, ytrain`.
2. Fit multiple `DecisionTreeRegressor` models (e.g., 500 trees) on these bootstrap samples.
3. Evaluate each tree's `R2` score and compare the average predictions of all trees (`mean` of their predictions) to a single tree's performance.
4. Reflect on how increasing or decreasing `max_depth` affects these results.

**Bootstrap sampling** provides a natural form of perturbation. By training trees on slightly different datasets, we reduce overfitting and improve our model's stability. The law of large numbers ensures that as we add more trees, our aggregated prediction becomes less variable.

**Main functions/methods:**

- `resample()`
- `DecisionTreeRegressor()`
- `r2_score()` (from `sklearn.metrics`)

---

## Section 4: Random Forests

*Introduction:*
**Random Forests** build on Bagging by injecting randomness not only in the data sampling but also in the feature selection at each split. This further decorrelates the trees, often leading to better performance than bagging alone.

**Task:**

1. Use `RandomForestRegressor()` from `sklearn.ensemble`.
2. Set `oob_score=True` to exploit the Out-Of-Bag estimates, a convenient built-in validation technique.
3. Experiment with different numbers of trees (`n_estimators`) and different proportions of features (`max_features`).
4. Identify the best combination based on the OOB score and confirm with the test set.
5. Examine `feature_importances_` to understand which predictors matter most.

**Out-of-Bag (OOB)** scoring is a clever validation method that uses the data not included in each bootstrap sample as a built-in test set. Random Forests leverage this technique to guide hyperparameter tuning without separate cross-validation, saving time and computation.

**Main functions/methods:**

- `RandomForestRegressor()`
- Using the attribute `oob_score_`
- Accessing `feature_importances_`

---

## Section 5: Gradient Boosting with AdaBoost

*Introduction:*
**Boosting** trains models sequentially, each one learning from the errors of the previous. Unlike Bagging (which operates in parallel), Boosting can produce highly accurate models by giving each successive tree a chance to fix its predecessor's mistakes. The `AdaBoostRegressor` is a classic implementation that adjusts the weights of each sample to focus on the hardest-to-predict instances.

**Task:**

1. Use `AdaBoostRegressor()` with `DecisionTreeRegressor` as the base estimator.
2. Experiment with `n_estimators`, `learning_rate`, and `max_depth` to see how these parameters impact convergence and overfitting.
3. Observe how the model's performance changes over the sequence of boosting stages with `staged_predict()` or `staged_score()`.

**Learning rate** in Boosting acts as a "step size" in the functional space. A smaller learning rate may require more iterations but can lead to more stable and robust predictions. Tuning the interplay between depth, learning rate, and number of estimators is key to optimizing a boosted model.

**Main functions/methods:**

- `AdaBoostRegressor()`
- `staged_score()`, `staged_predict()`
- Adjusting `learning_rate` and `n_estimators`

---

## Section 6: Reflection and Discussion

*Introduction:*
Having explored Decision Trees, Bagging, Random Forests, and Boosting, it's time to think critically about what you've learned. The complexity of single trees can be mitigated by ensemble methods, and both Bagging and Random Forests have shown you how averaging can improve stability. Boosting provides a different, iterative approach that often yields excellent performance but must be tuned carefully.

**Questions for Students:**

1. How did your best single Decision Tree model compare to the Random Forest and Boosted models?
2. What insights did the OOB score provide compared to a traditional train-test split?
3. How do feature importances from the Random Forest help you understand the problem's predictive structure?