1. The method of random forests, developed by Leo Breiman and Adele Cutler, was designed to improve the prediction accuracy of CART (classification and regression trees). It works by building a large number of trees which each "votes" on the outcome to make a final prediction.

   A random forest builds many trees by randomly selecting a subset of observations and a subset of independent variables to use for each tree. We could not just run CART many times, because we would get the same tree every time. Instead, each tree is given a random subset of the independent variables from which it can select the splits, and a training set that is a *bagged* or *bootstrapped* sample of the full dataset. This means that the data used as the training data for each tree is selected randomly with replacement from the full dataset. As an example, if we have five data points in our training set, labeled $\{1, 2, 3, 4, 5\}$, we might use the five observations $\{3, 2, 4, 3, 1\}$ to build the first tree, the five observations $\{5, 1, 2, 3, 2\}$ to build the second tree, etc. Notice that some observations are repeated, and some are not included in certain training sets. This gives each tree a slightly different training set, and helps make each tree see different things in the data.

   Just like in CART, there are some parameters that needed to be selected for a random forest model. One is the number of trees to build, which is typically set to be a few hundred. The more trees that are built, the longer the model will take to build. But if not enough trees are built, the full potential of the model might not be realized. Another parameter is needed to control the size the trees. A popular choice is to set a maximum depth of a tree. Random forest models have been shown to be less sensitive to the parameters of the model, so it is typically sufficient to set the parameters to reasonable values.

   In this assignment, we will investigate how the method of random forests performs in contextual decision-making, using the same setup as in the lecture note "From Predictions to Prescriptions", except that we now set the size of the training set $n = 1000$ and the size of the testing set $T = 500$. The data is given by *train.csv* and *test.csv*.

   (a) In the lecture note, we use linear regression as the machine learning model to predict the demand for a given observation of the covariates in the predict-then-optimize (PTO) approach. Use the the random forest model instead to see how the result changes.

   (b) The method of random forests can also be used in the "integrated" approach as it also provides a weight function. Let M be the number of trees built in a random forest model. For each $m = 1, \ldots, M$, tree m generates a disjoint partition of the value region of the covariates. Then, we may construct a weight function, say w_m^{tree}, for tree m using the partition of the tree. In particular, let ψ_m denote the binning rule of tree m that maps x to the corresponding bin (or leaf). As explained in the lecture note, the weight w_m^{tree} can be written as

   $$w_m^{\text{tree}}(x, x_i) = \frac{\mathbb{I}(\psi_m(x) = \psi(x_i))}{\#\{j : \psi_m(x) = \psi_m(x_j)\}}.$$

   That is, for tree m, given a new observation of the covariates x, we give equal weights to the

observations x_i in the training set that share the same bin (or leaf) as x, and these weights sum to one. On the other hand, the observations x_i in the training set that do not share the same leaf as x are given weights 0.

Note that we can construct such a weight function for each tree in the random forest model. (These weight functions are different because the trees are different.) Recall that when we use the random forest model for a prediction task, the random forest takes a majority vote over the M trees to generate the final prediction. This means that each tree carries the weight in the final decision. We may apply the same idea here for constructing the weight function for the random forest. Namely, we simply take an average on the weights generated by each tree as the weight generated by the random forest, i.e.,

$$w^{\text{forest}}(x, x_i) = \frac{1}{M}\sum_{m=1}^{M} w_m^{\text{tree}}(x, x_i) = \frac{1}{M}\sum_{m=1}^{M} \frac{\mathbb{I}(\psi_m(x) = \psi(x_i))}{\#\{j : \psi_m(x) = \psi_m(x_j)\}}.$$

Use this weight function in the "integrated" approach to see how the result changes.

Hint. The binning rule of each tree of the random forest model is already implemented in the `scikit-learn`. Explore its documentation to find out.

(c) Compare the results in (a) and (b) to the "integrated" approach based on kernel regression, where the Gaussian kernel is used.

Note. To achieve a good performance, you should write your own codes and tune the hyperparameters with cross-validation. Students who correctly use Bayesian Optimization for tuning will receive a bounus (10 pts at most).

(d) Design a new weight function for the "integrated" approach.
Implement it and compare the result with (a) to (c).
Hint. You can have a try with local linear regression.

2. Consider the following two functions to be minimized:

$$f(x, y) = (x-5)^2 + 2(y+3)^2 + xy,$$
$$g(x, y) = (1-(y-3))^2 + 10\left((x+4) - (y-3)^2\right)^2.$$

In the following sub-questions, run a gradient descent algorithm with the initial point $(x, y) = (0, 2)$. For function f, $T = 10$ iterations are allowed, while $T = 100$ iterations are allowed for function g. Report the function value after each iteration and visualize the convergence process.

(a) Run with a fixed learning rate of $\eta = 0.05$ for function f and $\eta = 0.0015$ for function g.

(b) Run with any variant of gradient descent you want, such as stochastic gradient descent or gradient descent with a learning rate schedule.

Note. Try to get the smallest function value after the T iterations. The performance of your gradient descent algorithm will be graded.