1. **What is bias/variance tradeoff? Give a few examples of how it works and discuss.**

In the field of data science, we always want our models to accurately describe or predict the pattern behind massive data. What the term “accurate” means is nothing more than “low error rates”. In this field, other than human mistakes, we would usually encounter three types of prediction errors: random errors (i.e., noise), bias, and variance. Since the noise is irreducible and ineliminable, we want to minimize the other two and optimize our models with our best efforts. Therefore, we would like to talk about bias error, variance error, and the bias/variance tradeoff.

Let us consider why we need to know about bias variance tradeoff? Because when we are talking about model prediction, we always want our data to be more perfect. We want to have less over- and underfitting conditions. We want similar results when trained with similar data sets, making the data less inconsistent and more accurate in predicting. One of the main obstacles that often lead to deviation from qualified predictions is bias and variance errors. I will first introduce what bias and variance errors are before we understand that there will be a tradeoff for a model to minimize bias and variance.

Bias is an expectation of error in model predictions. It is mathematically the difference between the average predicted value and the “true” value which our model aims to predict. Bias occurs because a model makes certain assumptions when it trains on the data given; and when it is introduced to a testing data, those assumptions may not be correct. For example, when we make linear regression model representation for a problem, we are assuming that what we are predicting have linear relationships with their features. The linearity assumed in this model might not be right, resulting in bias errors. If a model over-generalizes and has high bias, it usually signifies that the model is under fitting. It comes with poor training data and oversimplification, leading high error on both training and test data.

Variance is the variability in the results predicted by our model. It quantifies the difference in model prediction when the dataset is changed. Contrary to bias, the variance occurs when fluctuations in data are taken into account. The model “over-learns” the training data so that it cannot produce accurate predictions when encountering new testing data, which is usually called “overfitting”. Considering a model that predicts an extremely complex relationship between the outcome and input features when the relationship would have been explained by a simple quadratic equation. If a model fails to generalize on new data and focuses too much on training data, it signifies high error rates on test data.

An ultimate goal of any model is to be built with low bias and low variance. However, similar to a tradeoff in model complexity (which is also the reason behind the tradeoff between bias and variance), it is found that bias and variance have inverse relationship, meaning that increasing one will decrease in the other. To illustrates the conflict when we try to minimize both bias and variance, imagine a bullseye target where we can plot a model’s prediction on. The bullseye represents the target/truth while the predictions get worse as we move away from the bullseye. A model with low bias and low variance predicts points that are centered. Having high variance and low bias means the predicted points are generally around the center but are highly variable (i.e., far from each other). If a model has high bias and low variance, the model is consistent. Though the predicted points are close to each other, the predicted values are far away from the target. A model with high bias and variance, on the other hand, is a disaster, with its predicted points being far away from the bullseye and being highly variable. Such a model is thus inconsistent. Therefore, to build a good model and to gain better predictive performance, we need to balance well between bias and variance.

Ideally, it would be a sweet dream if we have just infinite training data and a perfect algorithm. While in reality, what we have is very limited resource and an imperfect model. So how do we balance between bias and variance? Mathematically, we want to optimize a model’s ability to describe, which is improved as model complexity improves. If the model is getting more and more complex, the performance of the model on the test data set would show higher variance and lower bias. This is also true in the opposite way if the model complexity becomes lower and the model is too simple with very few parameters. So, if we draw a graph of the sum of variance and bias error on a plot of Error versus Model Complexity, we would be able to find an upward parabola-like curve. The optimal model complexity (i.e., the best balance with low bias and low variance) is at the turning point. However, in practice, calculating bias and variance is almost impossible. Finding the right balance has then becomes an iterative process. More specifically, we deal with what a model has been found to be: either underfitting or overfitting. For example, after the training starts, if there is a high training error, we lower high bias by measures such as training longer, training a more complex model, or obtaining more features. If there is a high cross-validation error, we try to lower variance by obtaining more data, decreasing number of features etc. We train the model with different combinations of features, hyperparameters, different data set for training and test to find the right combination until prediction accuracy is as close as possible on both train and test data. This is how a best possible model is created.

Overall, I think I have explained what bias and variance errors are and what their tradeoff is. This is definitely crucial for understanding the behavior of prediction models and for building a good predictive model.

1. **If the response variable has different shapes in different regions, what model can we use to accommodate this and why?**
2. **What is bagging? What are pros and cons of the bagging? Discuss the R and Python packages of bagging.**

Decision trees are a powerful predictive modeling technique. However, since trees can have very different results given different training data, they often suffer from high variance. Bagging, also known as Bootstrap Aggregation, is commonly recognized as a relatively simple method to solve such a problem and reduce variance error from a data set. It is especially helpful when, for example, the sample size is considerably small, or when the predictors are relatively unstable. Bagging satisfies our needs by first taking multiple random samples (with replacement) from training data set, and then using each of the samples to build a separate model and predictions for the test data set. Depending on what type of models the technique is used with (and mostly with classification models and regression models), the models and predictions are either averaged or voted based on accuracy to create a better final prediction value. To better understand what bagging is, we need to first know that bagging consists of two elements: bootstrapping and aggregation.

Bootstrapping is an inferential statistic resampling method which makes the evaluation of many statistics of complex models feasible. It creates multiple samples with replacement of original data, allowing us to predict the population distribution even from a single sample and to measure the CI and bootstrap statistics. Aggregation is a method that combines the outcome of all those samples to measure the most accurate statistics.

With bootstrapping and aggregation together, we have bagging. Basically, bagging is one of ensemble algorithms that can help us enlacing the accuracy and stability of models. But we need to know what ensemble learning is so that we can understand bagging from a bigger picture. Ensemble learning is a “methodology of training” instead of a particular method or algorithm. In our real life, we know that many hands make light work. This is exactly the core idea behind ensemble learning: without creating a new algorithm, it combines a group of weak models to form a powerful one. This is mainly achieved by using methods including bagging, boosting, and stacking. The key idea behind bagging can be compared to “democracy”—it creates many slightly different models and treats each of them equally. Each one of the models has one vote, and a final result is then elected through a “democratic ballot”. In most cases, this results in a smaller variance and avoid overfitting and irregularities in a model. One of the most famous application of bagging is random forest, which is an ensemble learning method that constructs a multitude of decision trees at training time. By implementing the method of bagging, the random forest returns its output by selecting the most frequent result given by each individual decision tree (in classification tasks, or an averaged prediction in regression tasks).

Now that we know, theoretically, bagging lowers variance without an impact on bias, improves accuracy, avoids overfitting, and stabilizes predictions, but it is not always a panacea for all kinds of problems and data. For example, if the random samples we draw from training set are extremely similar (that is, a relatively low variance), we may then expect extremely similar predictions derived from them, where implementing bagging might seem redundant. Even if it does improve predictive performance somehow, the extra work may only add levels after levels of additional iterations with little benefit and complicates the maintenance of modeling pipeline in production. This makes bagging computationally expensive, discouraging its usage under certain circumstances. Besides, it has shown that a resultant model can experience lots of bias because bagging can carry high bias when aggregating if the samples also have high bias, introducing a loss of interpretability of the model. In some applications such as random forest, it is also possible for the technique to generate overfitting conditions for certain classification and regression tasks with high noise.

There are many programming languages for machine learning that can fulfill the task of bagging, such as R and Python. R is an open-source language to which people have contributed from all around the world. One of its advantages is that it is easy to find a package pre-written by someone else and can be directly used in any program. We would need 5 packages in R for bagging, including caret, dplyr, rpart, ipred, and e1071. For example, if we wish to fit a bagged model in R, we simply use bagging function from the “ipred” library. After fitting, we can visualize the importance of the predictor variables by using varlmp function from the caret library to create a variable importance plot for the fitted model. Eventually, we can adjust the fitted bagged model and predict over new observations. In Python, it is also possible to import different libraries to provide implementations of ensembles. For instance, with the scikit-learn package, we can build our own bagging algorithm using BaggingRegressor or BaggingClassifier. If we take classification problems as an example, we can first create a synthetic binary classification problem with a number of examples and input features. Then we can evaluate the bagging algorithm on the dataset using repeated stratified k-fold cross-validation. We can also use the bagging model as a final model to predict on classification over a new row of data.

In an application, there is no such a way being necessarily better than the other because the programming language one should choose directly depends on the requirements of a specific problem, personal preference, or the context of activities we want to perform. As long as we are familiar with how bagging works, language choice is never a problem.

Dealing with missing values is one of the common tasks in data analysis. In R, we usually see NA or some other value representing missing values. Missing values happen when our dataset is not complete and when information may be unavailable. We usually handle missing data by identifying, recoding, or excluding.

We can test for missing values by using, for example, is.na() function, which will return a logical vector with TRUE in locations that contain missing data. We can also easily identify the location and the amount of NAs by using which() and sum() functions. We can also find the total missing values within each column of a data frame using colSums().

We can also use normal subsetting and assignment operations to recode missing data. For example, to replace missing values with the mean values in a vector, we can subset the vector to identify the missing values and assign these elements a value. If we choose to recode NAs in one data frame variable, we can subset for the values in the particular variable and assign a desired value.

Most functions will return an NA if we fail to exclude missing values from our data. To exclude missing values, we can use na.rm = TRUE. We can also subset the data to obtain complete observations with no missing values. One of the convenient ways to do so is to leverage the complete.cases() function, which returns a logical vector that can identify which rows are complete cases. Using what the function gives us, we can easily subset our data frame which can return all rows that has comple.cases() found to be TRUE. An alternative way to do so is to use na.omt() function to omit all rows containing missing values.

Missing data can also be a substantial problem when analyzing a dataset if the amount of missing values is large relatively to the size of our dataset. This requests us to know how to impute missing data so that we do not wipe out those potentially useful datapoints from the dataset. If data is missing completely at random, we usually hope the missing data is not too much, considering a maximum threshold of 5% of the total for a large dataset. Otherwise, leaving the feature/sample out might be a better choice. If the missing data is not missing at random but with a pattern, it becomes a more serious issue, in which case we want to check how the data was gathered and try to understand why the data is missing. In R coding, we can use functions such as md.pattern() in the mice package to look at missing data pattern, or we can use a visual representation obtained using the VIM package to help us understand the pattern. To impute the missing data when a pattern can be identified, we can use methods including the mice() function and complete the dataset using the complete() function. Whenever an imputation happens, we need to inspect how well the imputed data was produced. We can inspect the distribution between original and imputed data by plotting. By drawing a scatterplot, we would be able to visually inspect the performance of imputations.

In the real world, the data mostly we see at the first glance is unstructured. Knowing how to deal with missing data is important to make our data clean and tidy before we apply any algorithm to derive the insights.