1.1

Decision Tree	Random Forest
A decision tree is a tree-like model of decisions along with possible outcomes in a diagram.	A classification algorithm consisting of many decision trees combined to get a more accurate result as compared to a single tree.
There is always a scope for overfitting, caused due to the presence of variance.	Random forest algorithm avoids and prevents overfitting by using multiple trees.
The results are not accurate.	This gives accurate and precise results.

1.2

There are a number of differences between XGBoost over random forest, AdaBoost and gradient boosting and we write some of them:

XGBoost advantage over random forest:

i. oost straight away prunes the tree with a score called “Similarity score” before entering into the actual modeling purposes. It considers the “Gain” of a node as the difference between the similarity score of the node and the similarity score of the children. If the gain from a node is found to be minimal then it just stops constructing the tree to a greater depth which can overcome the challenge of overfitting to a great extend. Meanwhile, the Random forest might probably overfit the data if the majority of the trees in the forest are provided with similar samples. If the trees are completely grown ones then the model will collapse once the test data is introduced. Therefore, major consideration is given to distributing all the elementary units of the sample with approximately equal participation to all trees.

ii. XGBoost is a good option for unbalanced datasets but we cannot trust random forest in these types of cases. In applications like forgery or fraud detection, the classes will be almost certainly imbalanced where the number of authentic transactions will be huge when compared with unauthentic transactions. In XGBoost, when the model fails to predict the anomaly for the first time, it gives more preferences and weightage to it in the upcoming iterations thereby increasing its ability to predict the class with low participation; but we cannot assure that random forest will treat the class imbalance with a proper process.

iii. When the model is encountered with a categorical variable with a different number of classes then there lies a possibility that Random forest may give more preferences to the class with more participation.

iv. XGBoost may more preferable in situations like Poisson regression, rank regression, etc. This is because trees are derived by optimizing an objective function.

XGBoost advantage over AdaBoost: AdaBoost is not optimized for speed, therefore being significantly slower than XGBoost. XGBoost was developed to increase speed and performance, with introducing regularization parameters to reduce overfitting that successfully reduces variance.

XGBoost advantage over gradient boosting: XGBoost is a more regularized form of Gradient Boosting. XGBoost uses advanced regularization (L1 & L2), which improves model generalization capabilities. XGBoost delivers high performance as compared to Gradient Boosting. Its training is very fast and can be parallelized across clusters.

2.1

Random Forest

Step 1: Create a "bootstrapped" dataset: To create a bootstrapped dataset that is the same size as the original, we just randomly select samples from the original dataset. We are allowed to pick the same sample more than once. For example, we randomly select four samples and we get the following table. Note that the third and fourth row are the same.

Bootstrapped Dataset

Blocked Arteries	Weight	Exercise	Genetics	Has Diabetes
Yes	180	Yes	Yes	Yes
No	125	No	No	No
Yes	167	No	Yes	Yes
Yes	167	No	Yes	Yes

Step 2: Create a decision tree using the bootstrapped dataset, but only use a random subset of variables (or columns) at each step. In this example, we will only consider 2 variables (columns) at each step. In this case, imagine that we randomly selected "Exercise" and "Blocked Arteries" as candidates for the root node. Just for the sake of the example, assume that "Exercise" did the best job separating the samples. Then make the tree as usual, but only considering a random subset of variables at each step.

Now, go back to Step 1 and repeat: Make a new bootstrapped dataset and build a tree considering a subset of variables at each step. Ideally, you'd do this 100's of times.

Using a bootstrapped sample and considering only a subset of the variables at each step results in a wide variety of trees. The variety is what makes random forests more effective than individual decision trees.

Now, how do we use the random forest we just created? First we get a new patient with all the measurements (the first four columns) and we want to know if they have diabetes or not. We check this data on all the trees of the random forest and we calculate the majority vote, that is, if more of the trees say "Yes"/"No" our prediction will tell "Yes"/"No"

For a detailed answer, search "StatQuest: Random Forests Part 1 - Building, Using and Evaluating" and see the answer to this question☺

Adaboost:

First, we'll start with some data. We create a forest of stumps with Adaboost to predict if a patient has Diabetes. We will make these predictions on a patient's "Blocked Arteries", "Exercise" and "Genetics" status and their "Weight". The first thing we do is give each sample a weight that indicates how important it is to be correctly classified.

Blocked Arteries	Weight	Exercise	Genetics	Has Diabetes	Sample Weight
No	210	Yes	Yes	No	
No	125	No	No	No	
Yes	180	Yes	Yes	Yes	
Yes	167	No	Yes	Yes	

Note: the "Sample Weight" is different from "Weight" Column. At the start, all samples get the same weight:

Blocked Arteries	Weight	Exercise	Genetics	Has Diabetes	Sample Weight
No	210	Yes	Yes	No	1/4
No	125	No	No	No	1/4
Yes	180	Yes	Yes	Yes	1/4
Yes	167	No	Yes	Yes	1/4

And this makes the samples all equally important. However, after we make the first stump, these weights will change in order to guide how the next stump is created. Now we need to make the first stump in the forest. This is done by finding the variables "Blocked Arteries", "Exercise", "Genetics" and "Weight", that does the best job classifying the samples. Note that because all the weights are the same, we can ignore them right now. We start by seeing how well "Genetics" classifies the samples. Of the 3 samples with "Genetics", 2 were correctly classified as having Diabetes and 1 was incorrectly classified. Of the 1 sample without "Genetics", it was correctly classified. Now we do the same thing for the other three columns. Note that for "Weight" column, we should determine the best weight to separate the patients using Decision Trees. Then we calculate the Gini Index for the four stumps and the lowest Gini Index will be the first stump in the forest. Then we should determine how much say this stump will have in the final classification based on how well it classified the samples. The total error for a stump is sum of the weights associated with the incorrectly classified samples. Note that the total error for a stump will always be between 0, for a prefect stump, and 1 for a horrible stump. We use the total error to determine amount of say this stump has in the final classification with the following formula:

$$\text{Amount of Say} = \frac{1}{2}\log\left(\frac{1-Total\ Error}{Total\ Error}\right)$$

Note that the total error is equal to the sum of the weights for the incorrectly classified samples.

We do this for all the samples and find the Amount of Say for all of them. Now we know how the Sample Weights for the incorrectly classified samples are used to determine the Amount of Say each stump gets.

Now we need to learn how to modify the weights so that the next stump will take the errors the current stump made into account. Here, the sample weights are not equal. We increase the sample weight for the incorrectly classified samples to emphasize the need for the next stump to correctly classify it by increasing its sample weights. This is the formula we will use to increase the sample weight for the sample that was incorrectly classified:

$$\text{New Sample Weight} = \text{Sample Weight} \times e^{\text{amount of say}}$$

Then we need to decrease the sample weights for all of the correctly classified samples using the following formula:

$$\text{New Sample Weight} = \text{Sample Weight} \times e^{\text{-amount of say}}$$

Now that we got the New Weights, we normalize the New Sample Weights so that they will add up to 1 and we repeat the above steps.

For a detailed answer, search “StatQuest: AdaBoost, Clearly Explained” and see the answer to this question☺

2.2

The initial idea for dealing with missing data in this context is to make an initial guess that could be bad, then gradually refine the guess until it is (hopefully) a good guess.

Because this person did not have Diabetes, the initial guess, and possibly bad, guess for the blocked arteries value is just the most common value for “Blocked Arteries” found in the other samples that do not have Diabetes. Among the people that do not have Diabetes, “No” is the most common value for “Blocked arteries” it occurs in 2 out of 2 samples. So “No” is our initial guess.

Since weight is numeric, our initial guess will be the median value of the patients that did not have Diabetes. In this case, the median value is 167.5

Here’s our new dataset with the filled in missing values.

Blocked Arteries	Weight	Exercise	Genetics	Has Diabetes
No	210	Yes	Yes	No
No	125	No	No	No
Yes	180	Yes	Yes	Yes
NO	**167.5**	No	Yes	Yes

Now we want to refine these guesses. We do this by first determining which samples are similar to the one with missing data. So let’s talk about how to determine similarity.

Step 1: Build a random forest

Step 2: Run all of the data down all of the trees.

We'll start by running all of the data down the first tree. For example, the first and fourth sample both ended up at the same leaf node. That means they are similar.

We keep track of similar samples using a Proximity Matrix. The proximity matrix has a row for each sample and it has a column for each sample.

	1	2	3	4
1				
2				
3				
4				

Because sample 3 and sample 4 ended up in the same leaf node, we put a one in fourth row and third column. We also put a one in third row and fourth column.

	1	2	3	4
1				
2				
3				1
4			1	

We do the same for other trees and do the same. Ultimately, we run the data down all the trees and the proximity matrix fills in. Then we divide each proximity value by the total number of trees. For example, we reach to the following table:

	1	2	3	4
1		0.2	0.1	0.1
2	0.2		0.1	0.1
3	0.1	0.1		0.8
4	0.1	0.1	0.8	

Now we use the proximity values for sample 4 to make better guesses about the missing data. For "Blocked Arteries", we calculate weighted frequency of "Yes" and "No" using proximity values as the weights:

$$\text{The weighted frequency for "Yes"} = \frac{1}{3} \times The\ weight\ for\ "Yes"$$

Calculating for "Blocked Arteries" we get: Weight for no =0.6 and Weight for yes = 0.03

So we go with No

For weight we use the proximities to calculate a weighted average:

$$\text{Weighted Average} = (125\times0.1)+(180\times0.1)+(210\times0.8)=198.5$$

We do these steps until the missing values converge.

For a detailed answer, search “StatQuest: Random Forests Part 2: Missing data and clustering” and see the answer to this question☺

3. The first thing we do is calculate the average weight which is 71.2. The next thing we do is build a tree based on the errors from the first tree. The errors that the previous tree made are the differences between the observed weights and the predicted weights (71.2). Now we will build a tree using the first three columns to predict the Residual column which just made.

$$\text{Predicted Weight} = \text{Average Weight} + \text{Learning Rate} \times \text{Tree leaf}$$

Gradient Boost deals with this problem by using a Learning Rate to scale the contribution from the new tree. The Learning Rate is a value between zero and one. In this case we set a Learning Rate equal to 0.1

We repeat this steps

For a detailed answer, search “StatQuest: Gradient Boost Part 1 (of 4): Regression Main Ideas” and see the answer to this question☺