Q1) **R-squared or Residual Sum of Squares (RSS) which one of these two is a better measure of goodness of fit model in regression and why?**

Ans) Neither R-squared nor Residual Sum of Squares (RSS) is inherently better than the other as a measure of goodness of fit for a regression model. Instead, they provide different perspectives on the model's performance.

R-squared represents the proportion of variance in the dependent variable that is explained by the independent variables. It ranges from 0 to 1, with higher values indicating a better fit. However, R-squared has some limitations. For instance, it always increases as more predictors are added to the model, even if some of them are irrelevant or redundant.

On the other hand, the RSS measures the total squared difference between the observed and predicated values of the dependent variable. It does not have an upper limit and generally decreases as more predictors are added to the model. However, a smaller RSS does not necessarily mean a better model if the number of predictors is increased at the expense of overfitting the data.

Therefore, it is essential to consider both R-squared and RSS, along with other diagnostic measures, to evaluate the overall performance of a regression model. Additionally, it is crucial to perform model selection techniques, such as stepwise regression, cross-validation, or information criteria (e.g., AIC or BIC), to find the optimal balance between model complexity and goodness of fit.

Q.2) What are TSS (Total Sum of Squares), ESS (Explained Sum of Squares) and RSS (Residual Sum of Squares) in regression. Also mention the equation relating these three metrics with each other.

Ans)

- Total Sum of Squares (TSS) measures the total variation in the response variable, y. It is calculated as the sum of the squared differences between each observed y-value and the overall mean of y.
- Explained Sum of Squares (ESS) measures the variation in y that is explained by the regression model. It is calculated as the sum of the squared differences between each predicted y-value and the overall mean of y.
- Res0idual Sum of Squares (RSS) measures the variation in y that is not explained by the regression model. It is calculated as the sum of the squared differences between each observed y – value and its corresponding predicted y – value.

These three metrices are related through the following equation:

TSS = ESS + RSS

This equation simply states that the total variation in y can be decomposed into the variation that is explained by the regression model and the variation that is not explained by the regression model.

Q.3) What is the need of regularization in machine learning?

Ans) Regularization is a technique used in machine learning to prevent overfitting, which occurs when a model learns the training data too well and performs poorly on unseen data =. Overfitting can be caused by a model having too many parameters relative to the number of observations, leading to the model memorizing the training data rather than learning the underlying patterns.

Regularization introduces a penalty term to the loss function, which discourages the model from assigning too much importance to any single feature. This has the effect of shrinking the coefficients of the features towards zero, which can help to reduce overfitting and improve the model's ability to generalize to new data.

Q.4) What is Gini–impurity index?

Ans) The Gini-impurity index is a measure of the impurity or heterogeneity of a set of data. It is commonly used in decision tree algorithms, such as the Classification and Regression Tree (CART) algorithm, to determine the optimal split at each node of the tree.

The Gini impurity index is defined as:

$$Gini = 1 - \sum_{i=1}^{C} (p_i)^2$$

Where p_i is the proportion of observations in the set that belong to class i.

The Gini impurity index ranges from 0 to 1, with 0 indicating that all observations in the set belong to the same class and 1 indicating that the observations are perfectly mixed across all classes.

In a decision tree, the goal is to find the split that minimizes the weighted sum of the Gini impurity indices of the child nodes. This is because a split that results in child nodes with low impurity is more likely to produce accurate predictions.

Q.5) Are unregularized decision-trees prone to overfitting? If yes, why?

Ans) Yes, unregularized decision trees can be prone to overfitting, especially when the tree is deep and has many nodes.

Overfitting occurs when a model is too complex and fits the training data too closely, capturing not only the underlying patterns but also the random noise in the data. This results in a model that performs well on the training data but poorly on new, unseen data.

In the case of decision trees, overfitting can occur when the tree is allowed to grow too deep and split the data into many small, homogeneous subsets. While this can result in a low training error, it can also lead to a high test error due to the tree's sensitivity to the random noise in the training data.

Regularization techniques, such as pruning, can be used to prevent overfitting in decision trees. Pruning involves removing nodes from the tree that do not contribute significantly to the model's performance. This can help to reduce the complexity of the tree and improve its ability to generalize to new data.

Here is an example of how to prune a decision tree in R using the 'rpart' package:

```
Library(rpart)

#load the iris dataset

Data(iris)

#Split the data into training and testing sets

Set.seed(123)

Train.indices <- sample(1:nrow(iris), nrow(iris) * 0.7)

Train_data <- iris[train_indices,]

Train_data <- iris[-train_indices,]


#Create a decision tree model with a maximum depth of 5

Tree_model <- rpart(Species ~ ., data = train_data, maxdepth = 5)


#Prune the tree to the optimal size using cost complexity pruning

Pruned_tree_model <- prune(tree_model, cp = tree_model $
cptable[which.min(tree_model$cptable[,"xerror"]),"cp])

#Predict the species of the test data using the pruned tree model

Predictions <- predict(pruned_tree_model, newdata = test_data, type = "class")

#calculate the accuracy of the predictions

Accuracy <- sum(pred)
```

Q.6) What is an ensemble technique in machine learning?

Ans) An ensemble technique in machine learning is a method that combines the predictions of multiple models to produce improved results. There are several ensemble techniques, including voting, stacking, bagging, and boosting.

Here's an example of stacking in Python using the scikit-learn library:

```
From sklearn.datasets import load_iris
From sklearn.model_selection import train_test_split
From sklearn.linear _model import LogisticRegression
From sklearn.ensemble import RandomForestClassifier
From sklearn.ensemble import StackingClassifier
#Load iris dataset
Iris = load_iris()
X = iris.data
Y = iris.target
#Split dataset into training and test sets
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.2, random_state = 42)
```

```
#Define base models
Log_reg = LogisticRegression(random_state = 42)

Rand_forest = RandomForestClassifier(random_state = 42)

#Define stacking model

Stacking_model = stckingClassifier(estimators = ['lr', log_reg), ('rf', rand_forest)],

Final_estimator = LogisticRegression()

)

#Train stacking model

Stacking_model.fit(X_train, Y_train)

#Predict on test set

Y_pred = stacking_model.predic(X_test)
```

In this example, we load the iris dataset and split it into training and test sets. We then define two base models, logistic regression, and random forest, and use them to train a stacking model. The stacking model combines the predictions of the base models using another logistic regression model. Finally, we predict on the test set using the stacking model.

Q.7) What is the difference between Bagging and Boosting techniques?

Ans) Bagging and Boosting are both ensemble methods used in machine learning, but they have some key differences.

Bagging, or Boosting Aggregating, is a method that involves creating multiple subsets of the original data, training a model on each subset, and then combining the predictions of each model to produce a final prediction. This helps to reduce the variance of the model and prevent overfitting. In Bagging, the models are built in parallel, and each model is given equal weight in the final prediction.

Boosting, on the other hand, is a method that involves building a sequence of models, with each model being trained to correct the errors made by the previous model. The final prediction is then made by combining the predictions of each model, with later models being given more weight in the final prediction. This helps to improve the bias of the model and increase its accuracy. In Boosting, the models are built sequentially, and each model is adapted based on the performance of the previous model.

To summarize, Bagging is a method that reduces variance and prevents overfitting by building multiple models in parallel and combining their predictions, while Boosting is a method that improves bias and increases accuracy by building a sequence of models that correct the errors of the previous models.

Q.8) What is out-of-bag error in random forests?

Ans) Out-of-bag (OOB) error in random forests is a method of measuring the prediction error. It is calculated using predictions from the trees that do not contain a particular sample in their respective bootstrap sample. This allows the RandomForestClassifier to be fit and validated while being trained. The OOB error is the average error for each sample calculated using predictions from the trees that do not contain that sample in their bootstrap sample. This method provides approximately the same model performance measurements as the test set and might be even more convenient than simple train_test validation because it doesn’t require dividing the data into training and test sets.

In other words, OOB error is a measure of how well a random forest model generalizes to unseen data. It is calculated by leaving out a subset of the training data when building each tree in the forest, and then using that left-out data to test the tree’s predictions. The OOB error is then the average error across all the trees in the forest.

It is important to note that OOB error is not a true measure of a model’s performance on new data, as it is calculated using the same data that was used to train the model. However, it can be a useful indicator of how well the model is likely to perform on new data, and can be used to tune the model’s hyperparameters.

Here is an example of how to calculate OOB error using scikit-learn library:

```
From sklearn.ensemble import RandomForestClassifier

#create a random forest classifier with 100 trees

Clf = RandomForestClassifier(n_estimators = 100, oob_score = True)

#fit the classifier to the data

Clf.fit(X,y)

#print the out-of-bag error

Print(clf.oob_score_)
```

In this example, ‘oob_score’ parameter is set to ‘True’ when creating the ‘RandomForestClassifier’ object. This tells scikit-learn to calculate the OOB error when fitting the classifier to the data. The ‘oob_score_’ attribute of the fitted classifier then contains the OOB error.

Q.9) What is K-fold cross-validation?

Ans) K-fold cross validation is a popular technique for assessing the performance and generalization error of machine learning models. It is commonly used to split a dataset into a training set and a test set, in order to evaluate the model’s ability to predict new, unseen data.

In K-fold cross validation, the original dataset is divided into K equal-sized subsets, or “folds”. The model is then trained on K-1 of these folds, while the remaining fold is held back as a validation set. This process is repeated K times, with a different fold being used as the validation set each time. The performance of the model is then averaged across all K runs to give an overall measure of its accuracy.

The main advantage of K-fold cross validation is that it provides a more robust estimate of the model’s performance than simply splitting the dataset into a single training and test set. This is because it allows the model to be trained and tested on different subsets of the data, reducing the risk of overfitting or underfitting to any particular subset.

K-fold cross validation is widely used in machine learning research and practice, and is often used as a standard benchmark for comparing the performance of different models. It is particularly useful when working with small or medium-sized datasets, where the risk of overfitting is higher.

Here's an example of how K-fold cross validation might be implemented in Python using the scikit-learn library:

```
From sklearn.model_selection import KFold

From sklearn.linear_model import LogisticRegression

From sklearn.metrics import accuracy_score

#Load the dataset

X = ….

Y = …..

#create a K-fold cross-validator

Kf = KFold(n_splits=5)

#Initialize an empty list to store the accuracy scores

Score = []

#Iterate over the folds

For train_index, test_index in kf.split(X):

#Split the data into training and test sets

X_train, X_test = X[train_index], X[test_index]

Y_train, y_test = y[train_index], y[test_index]

#Train a logistic regression model on the training set

Model = Logistic
```

Q.10) What is hyper parameter tuning in machine learning and why it is done?

Ans) Hyperparameter tuning in machine learning is the process of selecting the optimal values for a model's hyperparameters. Hyperparameters are settings that control the learning process of the model, such as the learning rate, the number of neurons in a neural network, or the kernel size in a support vector machine. The goal of hyperparameter tuning is to find the values that lead to the best performance on a given task.

Here's an example of hyperparameter tuning using GridSearchCV in scikit-learn:

```
From sklearn.datasets import load_iris

From sklearn.model_selection import GridSearchCV

From sklearn.tree import DecisionTreeClassifier
```

```
#Load iris dataset
Iris = load_iris()
X = iris.data
Y = iris.target
#Initialize Decision Tree Classifier
Clf = DecisionTreeClassifier()
#Define hyperparameters to tune
Param_grid = {‘max_depth’ : [None, 10, 20, 30],
              ‘min_samples_leaf’: [1, 2, 4],
              ‘max_features’: [2, 4, 8]}
#Create GridsearchCV object
Grid_search = GridSearchCV(clf, param_grid, cv=5, scoring=’accuracy’)
#Fit GridSearchCV object to the data
Grid_search.fit(X,y)
#print the best hyperparameters and the best score
Print(‘Tuned Decision Tree parameters:’, grid_search.best_params_)
Print(‘Best score is’, grid_search.best_score_)
Output:
Tuned Decision Tree parameters: {‘max_depth’: None, ‘min_samples_leaf’: 1, ‘max_features’: 8}
Best score is 0.9733333333333333333333
```

Hyperparameter tuning is done to improve the performance of a machine learning model. By selecting the optimal hyperparameters, we can improve the model’s accuracy, generalization, and other metrics. Hyperparameter tuning can also prevent overfitting and reduce training time.

Q.11) What issues can occur if we have a large learning rate in Gradient Descent?

Ans) If we have a large learning rate in Gradient Descent, several issues can occur:

1) Unstable Training: A large learning rate can cause the model to overshoot the optimal weights during the update step. This can lead to oscillations or divergence in the loss function, resulting in unstable training.
2) Slow Convergence: If the learning rate is too large, the model may take a long time to converge to the optimal weights. This is because the model will overshoot the optimal weights during each update step, leading to a zigzag pattern of weight updates.
3) Overfitting: A large learning rate can also lead to overfitting. This is because the model will be too sensitive to small fluctuations in the loss function during the update step. As a result, the model will learn the noise in the training data rather than the underlying patterns.

4) Incorrect Minima: If the learning rate is too large, the model may converge to a local minima rather than the global minima of the loss function. This can result in poor generalization performance on unseen data.

To avoid these issues, it is important to choose an appropriate learning rate for the problem at hand. This can be done through techniques such as grid search, random search, or Beyesian optimization. Additionally, using learning rate schedules or early stopping can help mitigate the impact of a large learning rate.

Q.12) Can we use Logistic Regression for classification of Non-Linear Data? If not, why?

Ans) Logistic Regression is a linear model, which means it can only model linear decision boundaries. Therefore, it may not be suitable for classification of non-linear data.

Non-linear data refers to data that cannot be separated by a straight line or a hyperplane. In such cases, a non-linear model is required to model the decision boundary.

However, there are ways to extend Logistic Regression to handle non-linear data. One such way is to use a technique called Kernel Trick. The Kernel Trick involves transforming the data into a higher dimensional space where it can be separated by a linear boundary. This is achieved by applying a kernel function to the data, which maps the data into a higher dimensional space.

For example, in the case of Logistic Regression, we can use a kernel function such as the Radial Basis Function (RBF) kernel to transform the data into a higher dimensional space. This results in a non-linear decision boundary in the original feature space.

Another way to handle non-linear data with Logistic Regression is to use a technique called Polynomial Regression. Polynomial Regression involves adding polynomial terms to the Logistic Regression model. This can help model non-linear decision boundaries.

In summary, while Logistic Regression is a linear model, it can be extended to handle non-linear data using techniques such as the Kernel Trick or Polynomial Regression. However, it may not be the best choice for non-linear classification tasks, and other non-linear models such as Support Vector Machines or Neural Networks may be more appropriate.

Q.13) Differentiate between Adaboost and Gradient Boosting.

Ans) Adaboost and Gradient Boosting are both ensemble methods used for classification and regression tasks. While they share some similarities, there are several key differences the two:

1) **Base Learner:** Adaboost uses a weak learner, typically a decision tree with a single split, as its base learner. Gradient Boosting, on the other hand, can use any differentiable loss function and any base learner, such as decision trees, linear models, or neural networks.
2) **Weight Updates:** In Adaboost, the weights of the training instances are updated after each iteration based on the error of the previous iteration. The weights of the misclassified instances are increased, and the weights of the correctly classified instances are decreased. In Gradient Boosting, the weights of the training instances are updated based on the gradient of the loss function.

3) **Loss Function:** Adaboost uses the exponential loss function, which is sensitive to misclassified instances. Gradient Boosting can use any differentiable loss function, such as squared error loss, absolute error loss, or logistic loss.
4) **Regularization:** Gradient Boosting has built-in regularization techniques, such as shrinking and subsampling, to prevent overfitting. Adaboost does not have built-in regularization techniques but can be regularized by early stopping or by limiting the number of iterations.
5) **Computational Complexity:** Gradient Boosting is generally more computationally expensive than Adaboost, as it requires computing the gradient of the loss function and updating the weights of the training instances after each iteration. Adaboost, on the other hand, only requires updating the weights of the training instances based on the error of the previous iteration.
6) **Performance:** Gradient Boosting generally better than Adaboost on complex datasets, as it can use more sophisticated base learners and regularization techniques. However, Adaboost can still perform well on simple datasets, and is often used as a baseline model for comparison.

In summary, Adaboost and Gradient Boosting are both ensemble methods used for classification and regression tasks. Adaboost uses a weak learner and the exponential loss function, while Gradient Boosting can use any differentiable.

Q.14) What is bias-variance trade off in machine learning?

Ans) The bias-variance tradeoff is a fundamental concept in machine learning that refers to the balance between a model's ability to minimize bias and variance. Bias is the error introduced by approximately a real-world problem, which may be extremely complicated, with a simplified model. Variance, on the other hand, is the error introduced by sensitivity to small fluctuations in the training set.

In machine learning, we aim to find the right balance between bias and variance to achieve optimal performance. A model with high bias and low variance will underfit the data, meaning that it has a high error rate for both training and testing data. In contrast, a model with low bias and high variance will overfit the data, meaning that it has a low error rate for training data but a high error rate for testing data.

Here is an example code that demonstrates the bias-variance tradeoff using polynomial regression:

```
Import numpy as np

From sklearn.model_selection import train_test_split

From sklearn.metrics import mean_squared_error

#Generate some data

Np.random.seed(42)

x = np.random.rand(100)

y = np.sin(2* np.pi * x) + np.random.rand(100)
```

```
#split the data into training and testing sets
  x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2)
#Define a function to compute the bias and variance of a model
Def bias_variance_decompasition(y_true, y_pred):
     bias = np.mean((np.mean(y_pred) – y_true) ** 2)
     variance = np.mean(((np.mean(y_pred, axis=0) – y_pred) **2))
#Define a function to fit polynomial regression models with different degrees
def fit_polynomial_regression(x_train, y_train, degree):
     model = np.poly1d(np.polyfit(x_train, y_train, degree))
     y_pred = model(x_test)
     return y_pred
```

Q.15) Give short description each of Linear, RBF, Polynomial kernels used in SVM.

Ans) In Support Vector machines (SVM), a kernel function is used to transform the input data into a higher dimensional space, where it is easier to find a hyperplane that separates the data into different classes. There are several commonly used kernel functions in SVM, including:

1) Linear kernel: The linear kernel is the simplest kernel function used in SVM. It calculates the dot product of two input vectors and adds an intercept term. The linear kernel is used when the data is already linearly separable in the input space. Here's an example code for the linear kernel:
   ```
   Def linear_kernel(x, y):
             Return np.dot(x, y) + 1
   ```

2) Radial Basis Function(RBF) Kernel: The RBF kernel is a popular choice for SVM because it can handle non-linearly separable data by mapping the input data into an infinite-dimensional space. The RBF kernel calculates the Education distance between two input vectors and exponentiates it with a parameter gamma. Here's an example code for the RBF kernel:

   ```
   def rbf_kernel(x, y, gamma = 1.0)
             return np.exp(-gamma * np.linalg.norm(x – y) ** 2)
   ```

3) Polynomial kernel: The polynomial kernel is used when the relationship between the input variables is polynomial. The polynomial kernel calculates the dot product of two input vectors raised to a power p, and multiplied by a constant c. Here's an example code for the polynomial kernel:

```
def poly_kernel(x, y, p=2, c=1.0)
        return(c + np.dot(x, y)) ** p
```

These kernel functions can be used in SVM to find a hyperplane that separates the data into different classes. The choice of kernel function depends on the nature of the data and the problem at hand.