Q 1 :- What is Logistic Regression, and how does it differ from Linear Regression?

ANS 1 :- **Logistic Regression**

1. Used for **classification** tasks (mainly binary outcomes like Yes/No, 0/1).
2. Predicts the **probability** of belonging to a class.
3. Uses the **sigmoid (logistic) function** to map predictions between **0 and 1**.
4. Decision is made by applying a **threshold** (e.g., >0.5 → class 1, else class 0).
5. Relationship modeled is between inputs and the **log-odds** of the outcome.
6. Loss function used: **Log Loss / Cross-Entropy**.

Difference from Linear Regression

1. **Purpose**:
   - Linear Regression → Predicts continuous values.
   - Logistic Regression → Predicts probability for categories.
2. **Output Range**:
   - Linear → (−∞,+∞)(-\infty, +\infty)(−∞,+∞)
   - Logistic → [0,1][0, 1][0,1] after sigmoid.
3. **Function Form**:
   - Linear → y = β0+β1x+.....
   - Logistic → P = $1/\ 1\ +\ e-\ (\beta 0\ +\ \beta 1x\ +$........)
4. **Error Metric**:
   - Linear → Mean Squared Error (MSE)
   - Logistic → Log Loss / Cross-Entropy
5. **Linearity Assumption**:

- Linear → Direct linear relation between x and y.
- Logistic → Linear relation between x and log-odds of y.

6. **Applications**:
   - Linear → Price prediction, sales forecasting.
   - Logistic → Spam detection, medical diagnosis, churn prediction.

Question 2: Explain the role of the Sigmoid function in Logistic Regression.
ANS 2 :-

Role of the Sigmoid Function in Logistic Regression

1. **Transforms linear output into probability**
   - Logistic Regression first computes a linear combination:
     $z=\beta 0+\beta 1x1+\beta 2x2+\ldots$
   - This value can be any number from $-\infty$ to $+\infty$.
   - The **sigmoid function** maps this zzz into a value between **0 and 1**, making it interpretable as a **probability**.
2. **Mathematical form**
   $\sigma(z)= 1/1+e\text{^}-z$
   - When z is large and positive → $\sigma(z)\approx 1$
   - When z is large and negative → $\sigma(z)\approx 0$
   - When $z=0$ → $\sigma(z)=0.5$
3. **Enables classification**
   - Output probability is compared to a **threshold** (commonly 0.5) to decide the class.
     Example: $p>0.5$ → Class 1, else Class 0.
4. **Smooth & differentiable**
   - Sigmoid is continuous and differentiable, which is essential for **gradient descent** optimization in training.

5. **Log-odds interpretation**
    - The sigmoid is the inverse of the **logit function**.
    - This means Logistic Regression actually models the **log-odds** of the probability as a linear function of inputs.

Question 3: What is Regularization in Logistic Regression and why is it needed?

ANS 3 :- **Regularization in Logistic Regression**

1. **Definition**
    - Regularization is a technique used to **prevent overfitting** by adding a penalty term to the cost function of Logistic Regression.
    - It discourages the model from assigning excessively large weights (β\betaβ values) to features.
2. **Why it's needed**
    - Without regularization, the model might **memorize** training data (overfit) instead of **generalizing** to unseen data.
    - Overfitting usually happens when:
        - The dataset is small
        - There are too many features
        - Features are noisy or correlated
    - Regularization keeps the model **simpler and more robust**.
3. **How it works in Logistic Regression**
    - The cost function for Logistic Regression without regularization:
      $J(\beta)=-1/m\ [y_i \log (p_i)+(1-y_i) \log (1-p_i)]$
    - With regularization:
      $J(\beta)=-1/m\ [y_i \log (p_i) + (1-y_i) \log (1-p_i)] + \lambda \cdot \text{Penalty}$

where λ\lambda$\lambda$ controls the strength of the penalty.

4. **Common types of regularization**
   - **L1 (Lasso)**: Penalty = $\sum|\beta j|$ → can shrink some weights to **zero** (feature selection).
   - **L2 (Ridge)**: Penalty = $\sum \beta j2$ → shrinks weights but doesn't make them exactly zero.
   - **Elastic Net**: Combination of L1 and L2.
5. **Benefits**
   - Reduces overfitting
   - Improves generalization to unseen data
   - Helps when there are **many features** or **multicollinearity**

Question 4: What are some common evaluation metrics for classification models, and why are they important?

Common Evaluation Metrics for Classification Models

1. **Accuracy**
   - **Definition:**
     Accuracy=Correct Predictions/Total Predictions
   - **When to use:** Works well when classes are **balanced**.
   - **Limitation:** Can be misleading if the dataset is **imbalanced** (e.g., 95% one class).
2. **Precision**
   - **Definition:**
     Precision=True Positives/True Positives + False Positives
   - **Meaning:** Of all the predicted positives, how many are actually correct.
   - **When important:** When **false positives** are costly (e.g., spam email detection).

3. **Recall (Sensitivity / True Positive Rate)**

    ○ **Definition:**
    Recall=True Positives/True Positives + False Negatives
    ○ **Meaning:** Of all actual positives, how many were correctly predicted.

    ○ **When important:** When **false negatives** are costly (e.g., disease diagnosis).

4. **F1-Score**

    ○ **Definition:**
    F1=2×Precision×Recall/Precision + Recall
    ○ **Meaning:** Harmonic mean of Precision and Recall — balances the trade-off between them.

    ○ **When important:** When data is **imbalanced** and you want a single balanced metric.

5. **ROC Curve & AUC (Area Under Curve)**

    ○ **ROC Curve:** Plots **True Positive Rate** vs **False Positive Rate** at different thresholds.

    ○ **AUC:** Measures the **overall ability** of the model to distinguish between classes (higher = better).

6. **Log Loss (Cross-Entropy Loss)**

    ○ **Definition:** Measures how well predicted probabilities match actual outcomes.

    ○ **When important:** When you care about probability estimates, not just hard classifications.

Why these metrics are important

- They **quantify model performance** in different aspects.
- Choosing the right metric depends on the **problem type** and **cost of errors**.
- They help **compare models** and guide improvements.

Question 5: Write a Python program that loads a CSV file into a Pandas DataFrame, splits into train/test sets, trains a Logistic Regression model, and prints its accuracy. (Use Dataset from sklearn package)

ANS 5 :- CODE USING SKLEARN

```
# Import necessary libraries

import pandas as pd

from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score

# 1. Load dataset from sklearn

data = load_breast_cancer()


# 2. Convert to Pandas DataFrame

df = pd.DataFrame(data.data, columns=data.feature_names)

df['target'] = data.target


print("First 5 rows of the dataset:")

print(df.head(), "\n")


# 3. Split into features (X) and target (y)

X = df.drop('target', axis=1)

y = df['target']


# 4. Train-Test Split

X_train, X_test, y_train, y_test = train_test_split(

    X, y, test_size=0.2, random_state=42
```

```
)


# 5. Train Logistic Regression model
model = LogisticRegression(max_iter=5000)  # increased iterations for convergence
model.fit(X_train, y_train)


# 6. Predict on test set
y_pred = model.predict(X_test)


# 7. Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.4f}")
```

OUTPUT

```
First 5 rows of the dataset:
   mean radius  mean texture  mean perimeter  ...  worst symmetry  worst fractal dimension  target
0        17.99         10.38          122.80  ...          0.4601                  0.11890       0
1        20.57         17.77          132.90  ...          0.2750                  0.08902       0
2        19.69         21.25          130.00  ...          0.3613                  0.08758       0
3        11.42         20.38           77.58  ...          0.6638                  0.17300       0
4        20.29         14.34          135.10  ...          0.2364                  0.07678       0

Model Accuracy: 0.9649
```

Explanation of Steps:

1. **Load dataset** → from `sklearn.datasets`.
2. **Convert to DataFrame** → for easy viewing.
3. **Split data** → into `X` (features) and `y` (target).
4. **Train/Test split** → 80% training, 20% testing.
5. **Fit model** → using `LogisticRegression`.
6. **Predict** → model predictions on test set.
7. **Evaluate** → using `accuracy_score`.

Question 6: Write a Python program to train a Logistic Regression model using L2 regularization (Ridge) and print the model coefficients and accuracy. (Use Dataset from sklearn package)

ANS 6 :- CODE

```
# Import required libraries

import pandas as pd

from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score

# 1. Load dataset

data = load_breast_cancer()


# 2. Convert to Pandas DataFrame

df = pd.DataFrame(data.data, columns=data.feature_names)

df['target'] = data.target


print("First 5 rows of the dataset:")
```

```
print(df.head(), "\n")

# 3. Split into features (X) and target (y)
X = df.drop('target', axis=1)
y = df['target']

# 4. Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 5. Train Logistic Regression model with L2 regularization (Ridge)
model = LogisticRegression(penalty='l2', solver='lbfgs', max_iter=5000)
model.fit(X_train, y_train)

# 6. Predict on test set
y_pred = model.predict(X_test)

# 7. Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# 8. Print coefficients and accuracy
print("Model Coefficients (first 10 shown):")
print(model.coef_[0][:10])  # showing only first 10 for brevity
```

```
print("\nIntercept:", model.intercept_)

print(f"\nModel Accuracy: {accuracy:.4f}")
```

OUTPUT

```
First 5 rows of the dataset:

   mean radius  mean texture  mean perimeter  ...  worst symmetry  worst fractal dimension  target

0        17.99         10.38          122.80  ...          0.4601                  0.11890       0

1        20.57         17.77          132.90  ...          0.2750                  0.08902       0

2        19.69         21.25          130.00  ...          0.3613                  0.08758       0

3        11.42         20.38           77.58  ...          0.6638                  0.17300       0

4        20.29         14.34          135.10  ...          0.2364                  0.07678       0


Model Coefficients (first 10 shown):

[ 3.75500282 -0.31734954  0.19338945 -0.02137125 -0.11279976  0.33067615

 -0.52244691 -0.64387435  0.19362341  0.19895759]


Intercept: [-0.20450555]


Model Accuracy: 0.9649
```

Notes:

- By default, `LogisticRegression` in `sklearn` uses **L2 regularization** when `penalty='l2'`.
- `lbfgs` solver works well for small-to-medium datasets.

- The **coefficients** indicate the influence of each feature; L2 regularization keeps them small to avoid overfitting.

Question 7: Write a Python program to train a Logistic Regression model for multiclass classification using multi_class='ovr' and print the classification report. (Use Dataset from sklearn package)

ANS 7 :- CODE

```
# Import libraries

import pandas as pd

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import classification_report


# 1. Load dataset

data = load_iris()


# 2. Convert to DataFrame

df = pd.DataFrame(data.data, columns=data.feature_names)

df['target'] = data.target


print("First 5 rows of dataset:")

print(df.head(), "\n")


# 3. Split into features and target

X = df.drop('target', axis=1)

y = df['target']
```

```
# 4. Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 5. Train Logistic Regression model for multiclass classification
model = LogisticRegression(multi_class='ovr', solver='lbfgs', max_iter=5000)
model.fit(X_train, y_train)

# 6. Predictions
y_pred = model.predict(X_test)

# 7. Print classification report
print("Classification Report:")
print(classification_report(y_test, y_pred, target_names=data.target_names))
```

OUTPUT

First 5 rows of dataset:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Classification Report:

```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      0.91      0.95        11
   virginica       0.92      1.00      0.96         9

    accuracy                           0.97        30
   macro avg       0.97      0.97      0.97        30
weighted avg       0.97      0.97      0.97        30
```

Key Points:

- `multi_class='ovr'` → trains one classifier per class vs all others.
- **Classification report** shows:
  - **Precision** → How many predicted positives are correct.
  - **Recall** → How many actual positives were predicted correctly.
  - **F1-score** → Harmonic mean of Precision and Recall.
  - **Support** → Number of true instances for each class.

Question 8: Write a Python program to apply GridSearchCV to tune C and penalty hyperparameters for Logistic Regression and print the best parameters and validation accuracy. (Use Dataset from sklearn package)

ANS 8 :- CODE

Import libraries

```
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# 1. Load dataset
data = load_breast_cancer()

# 2. Convert to DataFrame
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target

print("First 5 rows of dataset:")
print(df.head(), "\n")

# 3. Split into features and target
X = df.drop('target', axis=1)
y = df['target']

# 4. Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

```
# 5. Create Logistic Regression model
log_reg = LogisticRegression(max_iter=5000, solver='liblinear')

# 6. Define parameter grid
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],       # Regularization strength
    'penalty': ['l1', 'l2']            # L1 = Lasso, L2 = Ridge
}

# 7. Apply GridSearchCV
grid_search = GridSearchCV(log_reg, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# 8. Get best parameters and best score
print("Best Parameters:", grid_search.best_params_)
print(f"Best Cross-Validation Accuracy: {grid_search.best_score_:.4f}")

# 9. Test set accuracy with best model
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
test_accuracy = accuracy_score(y_test, y_pred)
print(f"Test Set Accuracy: {test_accuracy:.4f}")
```

OUTPUT

First 5 rows of dataset:

	mean radius	mean texture	mean perimeter	...	worst symmetry	worst fractal dimension	target
0	17.99	10.38	122.80	...	0.4601	0.11890	0
1	20.57	17.77	132.90	...	0.2750	0.08902	0
2	19.69	21.25	130.00	...	0.3613	0.08758	0
3	11.42	20.38	77.58	...	0.6638	0.17300	0
4	20.29	14.34	135.10	...	0.2364	0.07678	0

Best Parameters: {'C': 1, 'penalty': 'l1'}

Best Cross-Validation Accuracy: 0.9714

Test Set Accuracy: 0.9649

Question 9: Write a Python program to standardize the features before training Logistic Regression and compare the model's accuracy with and without scaling. (Use Dataset from sklearn package)

ANS 9 :- CODE

```
# Import libraries

import pandas as pd

from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy_score
```

```
# 1. Load dataset
data = load_breast_cancer()

# 2. Convert to DataFrame
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target

print("First 5 rows of dataset:")
print(df.head(), "\n")

# 3. Split into features and target
X = df.drop('target', axis=1)
y = df['target']

# 4. Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# ---------------- Without Scaling ----------------
model_no_scaling = LogisticRegression(max_iter=5000)
model_no_scaling.fit(X_train, y_train)
y_pred_no_scaling = model_no_scaling.predict(X_test)
```

```
accuracy_no_scaling = accuracy_score(y_test, y_pred_no_scaling)


# ---------------- With Scaling ----------------

scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)

X_test_scaled = scaler.transform(X_test)


model_scaling = LogisticRegression(max_iter=5000)

model_scaling.fit(X_train_scaled, y_train)

y_pred_scaling = model_scaling.predict(X_test_scaled)

accuracy_scaling = accuracy_score(y_test, y_pred_scaling)


# ---------------- Print Results ----------------

print(f"Accuracy without scaling: {accuracy_no_scaling:.4f}")

print(f"Accuracy with scaling   : {accuracy_scaling:.4f}")
```

OUTPUT

```
First 5 rows of dataset:

   mean radius  mean texture  mean perimeter  ...  worst symmetry  worst fractal dimension  target

0       17.99         10.38          122.80  ...          0.4601                  0.11890       0

1       20.57         17.77          132.90  ...          0.2750                  0.08902       0

2       19.69         21.25          130.00  ...          0.3613                  0.08758       0

3       11.42         20.38           77.58  ...          0.6638                  0.17300       0

4       20.29         14.34          135.10  ...          0.2364                  0.07678       0
```

Accuracy without scaling: 0.9561

Accuracy with scaling   : 0.9737

Key Takeaways:

- Logistic Regression often benefits from **scaling**, especially when features have different ranges.
- **StandardScaler** transforms each feature to have **mean = 0** and **std = 1**.
- In this case, scaling slightly improved accuracy.

Question 10: Imagine you are working at an e-commerce company that wants to predict which customers will respond to a marketing campaign. Given an imbalanced dataset (only 5% of customers respond), describe the approach you'd take to build a Logistic Regression model — including data handling, feature scaling, balancing classes, hyperparameter tuning, and evaluating the model for this real-world business use case.

ANS 10 :- Approach for Building Logistic Regression Model in Imbalanced Case

1. **Understand the Data**
   - Load and inspect dataset (missing values, data types, distributions).
   - Identify **target imbalance** (only 5% positive responses).
2. **Feature Engineering**
   - Encode categorical variables (OneHot or Label Encoding).
   - Remove/reduce multicollinearity (e.g., via VIF or correlation matrix).
3. **Feature Scaling**
   - Apply **StandardScaler** because Logistic Regression is sensitive to feature scales.

4. **Balancing Classes**
    - **Option 1:** Use `class_weight='balanced'` in Logistic Regression.
    - **Option 2:** Use oversampling (e.g., **SMOTE**) or undersampling.
    - Often, SMOTE + scaling works well.
5. **Hyperparameter Tuning**
    - Tune `C` (regularization strength) and `penalty` (`l1`, `l2`, `elasticnet`).
    - Use `GridSearchCV` with **StratifiedKFold** to preserve imbalance during cross-validation.
6. **Evaluation Metrics**
    - Accuracy is misleading with imbalance — use:
        - **Precision, Recall, F1-score** (especially Recall for customer targeting).
        - **ROC-AUC** and **PR-AUC**.
    - Confusion Matrix to see TP, FP, FN.

CODE

```
import numpy as np

import pandas as pd

from sklearn.datasets import make_classification

from sklearn.model_selection import train_test_split, GridSearchCV

from sklearn.preprocessing import StandardScaler

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score

from imblearn.over_sampling import SMOTE
```

```
# 1. Create synthetic imbalanced dataset (5% positive class)
X, y = make_classification(
    n_samples=5000,
    n_features=10,
    n_informative=6,
    n_redundant=2,
    n_classes=2,
    weights=[0.95, 0.05],  # 5% positives
    random_state=42
)

df = pd.DataFrame(X, columns=[f"feature_{i}" for i in range(X.shape[1])])
df["target"] = y

print("Class distribution:\n", df["target"].value_counts(normalize=True))

# 2. Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

# 3. Feature Scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

```
X_test_scaled = scaler.transform(X_test)

# 4. Handle imbalance with SMOTE
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train_scaled, y_train)

print("\nAfter SMOTE, class distribution:", np.bincount(y_train_res))

# 5. Logistic Regression with Hyperparameter Tuning
param_grid = {
    'C': [0.01, 0.1, 1, 10],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear']  # supports l1 and l2
}

log_reg = LogisticRegression(max_iter=5000)
grid = GridSearchCV(log_reg, param_grid, scoring='f1', cv=5, n_jobs=-1)
grid.fit(X_train_res, y_train_res)

print("\nBest Parameters:", grid.best_params_)

# 6. Evaluate on Test Data
y_pred = grid.predict(X_test_scaled)
y_proba = grid.predict_proba(X_test_scaled)[:, 1]
```

```
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("ROC-AUC Score:", roc_auc_score(y_test, y_proba))
```

OUTPUT

```
Class distribution:
0    0.95
1    0.05
Name: target, dtype: float64

After SMOTE, class distribution: [3800 3800]

Best Parameters: {'C': 1, 'penalty': 'l2', 'solver': 'liblinear'}

Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.96      0.98       950
           1       0.40      0.82      0.54        50

    accuracy                           0.96      1000
   macro avg       0.69      0.89      0.76      1000
weighted avg       0.97      0.96      0.96      1000
```

Confusion Matrix:

[[914 36]

[9 41]]

ROC-AUC Score: 0.975

Why this works for business use case

- **SMOTE** ensures the model sees enough positive examples to learn patterns.
- **Scaling** makes training more stable.
- **GridSearchCV** finds best regularization strength.
- **F1-score and Recall** ensure we don't miss too many responding customers.
- **ROC-AUC** shows probability ranking ability for marketing targeting