ML ASSIGNMETS

Certainly! Below is the comprehensive explanation of all 7 assignments, following the format you've requested, with the algorithm description, code, working flow, and working of each algorithm step-by-step, including formulas where necessary.

---

1. Assignment on Linear Regression

1.1 Algorithm: Linear Regression

- **What it is**:

  Linear regression is a statistical method for modeling the relationship between a dependent variable (target) and one or more independent variables (features). The goal is to fit a linear equation to the observed data.

- **What it is used for**:

  Linear regression is used to predict a continuous value. It's commonly applied in areas like forecasting (e.g., predicting sales, stock prices) or estimating relationships between variables.

1.2 Code:

```
# Step 1: Import Libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Step 2: Load Dataset
data = pd.read_csv('linear_regression_data.csv')  # Loading d
ataset from CSV
print("Dataset Preview:\n", data.head())  # Display the first
few rows
```

```
# Step 3: Prepare Features and Target Variable
X = data.iloc[:, :-1].values  # Features (all columns except
the last one)
y = data.iloc[:, -1].values   # Target variable (the last col
umn)

# Step 4: Split Data into Training and Test Set
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=0.3, random_state=42)

# Step 5: Create Linear Regression Model
model = LinearRegression()  # Initializing the Linear Regress
ion model
model.fit(X_train, y_train)  # Fitting the model on training
data

# Step 6: Make Predictions
y_pred = model.predict(X_test)  # Predicting values using the
trained model

# Step 7: Evaluate Model
mse = mean_squared_error(y_test, y_pred)  # Calculate Mean Sq
uared Error
r2 = r2_score(y_test, y_pred)  # Calculate R-squared value

print(f"Mean Squared Error: {mse}")
print(f"R-squared value: {r2}")
```

1.3 Code Working Flow:

1. **Import Libraries**:
   We import necessary libraries for data manipulation (`pandas`), machine learning (`sklearn`), and evaluation metrics (`mean_squared_error`, `r2_score`).

2. **Load Dataset**:
   Load the dataset and display the first few rows to get a sense of the data.
3. **Prepare Data**:
   Separate features (X) and target variable (y).
   `X` contains the independent variables, and `y` is the dependent variable.
4. **Train-Test Split**:
   Split the dataset into training and testing sets (70% training, 30% testing).
5. **Create Model**:
   Create an instance of the
   `LinearRegression` model and fit it to the training data.
6. **Make Predictions**:
   Use the trained model to predict values for the test data.
7. **Evaluate Model**:
   Calculate performance metrics:
   Mean Squared Error (MSE) and **R-squared**.

1.4 Working of Linear Regression Algorithm:

1. **Formula**:

   Linear regression follows the equation:

   Y=b0+b1X1+b2X2+...+bnXnY = b_0 + b_1X_1 + b_2X_2 + ... + b_nX_n

   Where:

   - YY = Target variable
   - X1,X2,...,XnX_1, X_2, ..., X_n = Independent variables
   - b0b_0 = Intercept
   - b1,b2,...,bnb_1, b_2, ..., b_n = Coefficients (weights)

2. **Step-by-Step**:

   - The algorithm finds the optimal values for using a method called **Ordinary Least Squares (OLS)**.

     b0,b1,...,bnb_0, b_1, ..., b_n

- The coefficients are calculated by minimizing the **sum of squared residuals** (errors between predicted and actual values).

Cost function (MSE)=1m∑i=1m(yi−y^i)2\text{Cost function (MSE)} = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2

- Where mm is the number of data points, yiy_i is the actual value, and y^i\hat{y}_i is the predicted value.
- The algorithm minimizes the above cost function to find the best-fit line that predicts the target variable based on features.

---

2. Principal Component Analysis (PCA)

2.1 Algorithm: Principal Component Analysis (PCA)

- **What it is**:

  PCA is a dimensionality reduction technique that transforms high-dimensional data into a lower-dimensional form, while retaining as much variance as possible.

- **What it is used for**:

  PCA is commonly used in data preprocessing, feature extraction, and to reduce computational complexity without losing significant information.

2.2 Code:

```
# Step 1: Import Required Libraries
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Step 2: Load Dataset
data = pd.read_csv('pca_data.csv')
print("Dataset Preview:\n", data.head())
```

```
# Step 3: Preprocessing the Data (Scaling)
scaler = StandardScaler()  # Standardizing the data
data_scaled = scaler.fit_transform(data)

# Step 4: Apply PCA
pca = PCA(n_components=2)  # Reducing to 2 dimensions
pca_result = pca.fit_transform(data_scaled)

# Step 5: Explained Variance
print(f"Explained Variance Ratios: {pca.explained_variance_ra
tio_}")
```

2.3 Code Working Flow:

1. **Import Libraries**:
   We import necessary libraries for PCA (`PCA` from `sklearn.decomposition`), data scaling (`StandardScaler`), and data manipulation (`pandas` , `numpy`).
2. **Load Dataset**:
   Load the dataset to be reduced to fewer dimensions and preview it.
3. **Scale Data**:
   Scale the data to zero mean and unit variance using **StandardScaler**. PCA is sensitive to the scale of data.
4. **Apply PCA**:
   Apply PCA to reduce the dataset's dimensions (in this case, reducing to 2 principal components).
5. **Explained Variance**:
   Display the proportion of variance explained by each of the principal components.

2.4 Working of PCA Algorithm:

1. **Step 1: Standardization**

First, the dataset is standardized to ensure that features with different units do not dominate the results.

2. **Step 2: Covariance Matrix**

   Compute the covariance matrix of the data to understand how features vary together.

3. **Step 3: Eigenvalues and Eigenvectors**

   Compute the eigenvectors (principal components) and eigenvalues (the amount of variance each principal component explains).

4. **Step 4: Sort Eigenvalues**

   Sort the eigenvalues in decreasing order to determine the importance of each principal component.

5. **Step 5: Project Data**

   The data is projected onto the new feature space formed by the top principal components (those with the highest eigenvalues).

   X′=X·WX' = X \cdot W

   Where:

   - XX is the original data matrix.
   - WW is the matrix of eigenvectors.

---

3. Decision Tree Classifier

3.1 Algorithm: Decision Tree

- **What it is**:

  A decision tree is a non-linear classification and regression algorithm. It splits the data into branches based on feature values to make decisions or predictions.

- **What it is used for**:

  Decision trees are used for classification tasks where the goal is to predict a discrete label based on input features.

3.2 Code:

```
# Step 1: Import Required Libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_re
port
import matplotlib.pyplot as plt
from sklearn import tree

# Step 2: Load Dataset
data = pd.read_csv('DecisionTree_Data.csv')
print("Dataset Preview:\n", data.head())

# Step 3: Prepare Features and Target Variable
X = data.iloc[:, :-1].values  # Features
y = data.iloc[:, -1].values   # Target variable

# Step 4: Split Data into Training and Test Set
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=0.3, random_state=42)

# Step 5: Create Decision Tree Model
clf = DecisionTreeClassifier(random_state=42)
clf.fit(X_train, y_train)

# Step 6: Make Predictions
y_pred = clf.predict(X_test)

# Step 7: Evaluate Model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

```
# Step 8: Display Classification Report
print("Classification Report:\n", classification_report(y_tes
t, y_pred))
```

Step 9: Visualize the Decision Tree

plt.figure(figsize=(12, 8))
tree.plot_tree(clf, filled=True, feature_names=data.columns[:-1],
class_names=np.unique(y).astype(str), rounded=True)
plt.title('Decision Tree Visualization')
plt.show()

```
---

#### **3.3 Code Working Flow**:
1. **Import Libraries**:
   We import necessary libraries for decision tree modeling
(`DecisionTreeClassifier`) and visualization (`matplotlib`, `
sklearn.tree`).
2. **Load Dataset**:
   Load the dataset and prepare it for training and testing.
3. **Prepare Data**:
   Separate the dataset into features (X) and the target vari
able (y).
4. **Train-Test Split**:
   Split the data into training and testing sets.
5. **Create Model**:
   Create and train the decision tree model.
6. **Make Predictions**:
   Predict the test data labels using the trained model.
7. **Evaluate Model**:
   Evaluate the model's accuracy and print the classification
report.
```

```
8. **Visualize Decision Tree**:
   Visualize the decision tree to understand how decisions ar
e made.

---

#### **3.4 Working of Decision Tree Algorithm**:
1. **Step 1: Feature Selection**
   The decision tree chooses the best feature to split the da
ta at each node, based on certain criteria (like **Gini Impur
ity** or **Entropy**).

2. **Step 2: Split Data**
   The data is split recursively at each node, based on the f
eature that provides the best split.

   **Gini Impurity** is calculated as:
   \[
   Gini = 1 - \sum (p_i)^2
   \]
   Where \( p_i \) is the probability of class \( i \) at tha
t node.

3. **Step 3: Stopping Condition**
   The algorithm stops splitting when it reaches a stopping c
riterion, like the maximum depth of the tree or when all data
points belong to the same class.

---

---

### **4. Implement Naive Bayes Classification Algorithm**

---
```

````
#### **4.1 Algorithm: Naive Bayes**
- **What it is**:
  Naive Bayes is a probabilistic classification algorithm based on applying **Bayes' theorem** with the assumption of independence between the features.

- **What it is used for**:
  Naive Bayes is particularly useful for text classification tasks like spam detection, sentiment analysis, etc.

---

#### **4.2 Code**:

```python
# Step 1: Import Required Libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report

# Step 2: Load Dataset
data = pd.read_csv('naive_bayes_data.csv')
print("Dataset Preview:\n", data.head())

# Step 3: Prepare Features and Target Variable
X = data.iloc[:, :-1].values  # Features
y = data.iloc[:, -1].values   # Target variable

# Step 4: Split Data into Training and Test Set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Step 5: Create Naive Bayes Model
````
````

```
model = GaussianNB()
model.fit(X_train, y_train)

# Step 6: Make Predictions
y_pred = model.predict(X_test)

# Step 7: Evaluate Model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Step 8: Classification Report
print("Classification Report:\n", classification_report(y_tes
t, y_pred))
```

4.3 Code Working Flow:

1. **Import Libraries**:
   Import necessary libraries like
   `GaussianNB` for Naive Bayes classification, and metrics to evaluate the model.
2. **Load Dataset**:
   Load the dataset and prepare it for model training.
3. **Prepare Data**:
   Separate the data into features and target variable.
4. **Train-Test Split**:
   Split the dataset into training and testing sets.
5. **Create Model**:
   Create the Naive Bayes model and fit it to the training data.
6. **Make Predictions**:
   Predict the labels for the test data.
7. **Evaluate Model**:
   Calculate and print accuracy, classification report.

4.4 Working of Naive Bayes Algorithm:

1. **Bayes' Theorem**:

   Bayes' theorem is the foundation of the Naive Bayes algorithm, and it's used to calculate the probability of a class given the features:

   P(C ∣ X)=P(X ∣ C)P(C)P(X)P(C|X) = \frac{P(X|C) P(C)}{P(X)}

   Where:

   - P(C ∣ X)P(C|X) = Probability of class given the features

     CC

     XX

   - P(X ∣ C)P(X|C) = Likelihood of the features given class

     CC

   - P(C)P(C) = Prior probability of class

     CC

   - P(X)P(X) = Probability of features

     XX

2. **Assumption of Independence**:

   Naive Bayes assumes that all features are independent of each other, which simplifies the computation of P(X ∣ C)P(X|C) as:

   P(X ∣ C)=∏i=1nP(Xi ∣ C)P(X|C) = \prod_{i=1}^{n} P(X_i|C)

   This assumption of feature independence is why it's called "naive".

3. **Classification**:

   The class with the highest posterior probability is chosen as the predicted class. The class probabilities are computed for all classes, and the highest probability wins.

Note: Let me know if you would like to continue the rest of the algorithms or need further clarification!

5. SVM Classification

5.1 Algorithm: Support Vector Machine (SVM)

- **What it is**:

  Support Vector Machine (SVM) is a supervised machine learning algorithm that can be used for classification and regression tasks. SVM works by finding the hyperplane that best separates the classes in the feature space.

- **What it is used for**:

  SVM is widely used for binary classification problems, such as spam detection, image recognition, and bioinformatics (e.g., cancer classification).

5.2 Code:

```
# Step 1: Import Libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_re
port

# Step 2: Load Dataset
data = pd.read_csv('svm_data.csv')
print("Dataset Preview:\n", data.head())

# Step 3: Prepare Features and Target Variable
X = data.iloc[:, :-1].values  # Features
y = data.iloc[:, -1].values   # Target variable

# Step 4: Split Data into Training and Test Set
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=0.3, random_state=42)

# Step 5: Create SVM Model
model = SVC(kernel='linear')  # Using linear kernel
model.fit(X_train, y_train)
```

```
# Step 6: Make Predictions
y_pred = model.predict(X_test)

# Step 7: Evaluate Model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Step 8: Classification Report
print("Classification Report:\n", classification_report(y_tes
t, y_pred))
```

5.3 Code Working Flow:

1. **Import Libraries**:
   Import necessary libraries for SVM classification (`SVC` from `sklearn.svm`), data manipulation (`pandas`), and model evaluation (`accuracy_score`, `classification_report`).
2. **Load Dataset**:
   Load the dataset and preview it.
3. **Prepare Data**:
   Separate features and target variables.
4. **Train-Test Split**:
   Split the dataset into training and testing sets.
5. **Create Model**:
   Create an instance of the SVM model with a linear kernel, then train it with the training data.
6. **Make Predictions**:
   Predict the labels for the test set.
7. **Evaluate Model**:
   Calculate the accuracy of the model and print the classification report.

5.4 Working of SVM Algorithm:

1. **Objective**:

   The objective of SVM is to find a hyperplane that best divides the dataset into two classes. The margin between the hyperplane and the closest data points (called support vectors) is maximized.

2. **Kernel Trick**:

   SVM can operate in higher-dimensional spaces using the **kernel trick**, which transforms the data into a higher-dimensional space to find a linear hyperplane that separates the data. The most commonly used kernels are **linear**, **polynomial**, and **RBF**.

3. **Formula**:

   The decision function for the SVM classifier is given by:

   f(x)=wTx+bf(x) = w^T x + b

   Where:

   - ww is the weight vector (normal to the hyperplane)
   - xx is the feature vector
   - bb is the bias term

   The goal is to maximize the margin 1∥w∥\frac{1}{\|w\|} while correctly classifying the data points.

4. **Classification Decision**:

   The class of a new data point is decided based on the sign of f(x)f(x). If f(x)>0f(x) > 0, the point belongs to one class, and if f(x)<0f(x) < 0, it belongs to the other class.

---

6. K-Means Clustering

6.1 Algorithm: K-Means Clustering

- **What it is**:

  K-Means is an unsupervised machine learning algorithm used to partition a dataset into clusters. Each data point is assigned to the cluster with the nearest centroid.

- **What it is used for**:

  K-Means is used for clustering tasks, like customer segmentation, market research, or grouping similar data points.

6.2 Code:

```
# Step 1: Import Libraries
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Step 2: Load Dataset
data = pd.read_csv('kmeans_data.csv')
print("Dataset Preview:\n", data.head())

# Step 3: Prepare Data
X = data.values  # Convert the dataset to numpy array

# Step 4: Apply K-Means Clustering
kmeans = KMeans(n_clusters=3, random_state=42)  # Set number
of clusters
kmeans.fit(X)

# Step 5: Cluster Centers and Labels
centroids = kmeans.cluster_centers_  # Centroids of clusters
labels = kmeans.labels_  # Labels of clusters

# Step 6: Visualize the Clusters
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')  # Sc
atter plot of data points
plt.scatter(centroids[:, 0], centroids[:, 1], marker='X', col
or='red', s=200)  # Plot centroids
```

```
plt.title('K-Means Clustering')
plt.show()
```

6.3 Code Working Flow:

1. **Import Libraries**:
   Import necessary libraries for clustering (`KMeans` from `sklearn.cluster`), data handling (`pandas` , `numpy`), and visualization (`matplotlib.pyplot`).
2. **Load Dataset**:
   Load the dataset and preview it.
3. **Prepare Data**:
   Convert the dataset into a NumPy array for compatibility with the K-Means algorithm.
4. **Apply K-Means**:
   Apply the K-Means algorithm to the data, setting the number of clusters to 3 (or any other number based on the problem).
5. **Extract Results**:
   Get the centroids of the clusters and the labels (cluster assignments for each data point).
6. **Visualize Clusters**:
   Use a scatter plot to visualize the clustered data points and the cluster centroids.

6.4 Working of K-Means Algorithm:

1. **Step 1: Initialize Centroids**

   K initial centroids are chosen randomly from the data points.
2. **Step 2: Assign Data Points**

   Each data point is assigned to the nearest centroid, forming clusters.
3. **Step 3: Update Centroids**

The centroids are updated by calculating the mean of all the points assigned to each cluster.

4. **Step 4: Repeat**

   The steps of assigning points to clusters and updating centroids are repeated until convergence (when the centroids do not change significantly).

   The algorithm minimizes the following objective function:

   $$J = \sum_{i=1}^{k} \sum_{x_j \in C_i} \| x_j - \mu_i \|^2$$

   Where:

   - C_i is the set of points in cluster i
   - μ_i is the centroid of cluster i
   - x_j is the data point

---

7. Gradient Boosting Classifier

7.1 Algorithm: Gradient Boosting Classifier

- **What it is**:

  Gradient Boosting is an ensemble learning technique that builds a model in a stage-wise manner, where each new model corrects the errors of the previous ones.

- **What it is used for**:

  Gradient Boosting is used for both classification and regression tasks. It's effective in dealing with complex datasets and can handle overfitting better than other models.

7.2 Code:

```
# Step 1: Import Required Libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, classification_re
port

# Step 2: Load Dataset
data = pd.read_csv('gradient_boosting_data.csv')
print("Dataset Preview:\n", data.head())

# Step 3: Prepare Features and Target Variable
X = data.iloc[:, :-1].values  # Features
y = data.iloc[:, -1].values   # Target variable

# Step 4: Split Data into Training and Test Set
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=0.3, random_state=42)

# Step 5: Create Gradient Boosting Model
model = GradientBoostingClassifier(n_estimators=100, random_s
tate=42)
model.fit(X_train, y_train)

# Step 6: Make Predictions
y_pred = model.predict(X_test)

# Step 7: Evaluate Model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Step 8: Classification Report
```

```
print("Classification Report:\n", classification_report(y_tes
t, y_pred))
```

*7.

3 Code Working Flow**:

1. **Import Libraries**:
   Import necessary libraries for Gradient Boosting (`GradientBoostingClassifier`), data handling (`pandas` , `numpy`), and evaluation (`accuracy_score` , `classification_report`).
2. **Load Dataset**:
   Load the dataset and preview it.
3. **Prepare Data**:
   Separate the features and target variable.
4. **Train-Test Split**:
   Split the data into training and testing sets.
5. **Create Model**:
   Initialize the Gradient Boosting Classifier model with 100 estimators and fit it to the training data.
6. **Make Predictions**:
   Predict the test labels.
7. **Evaluate Model**:
   Evaluate the model's accuracy and print the classification report.

7.4 Working of Gradient Boosting Algorithm:

1. **Step 1: Initialize a Model**

   Start with a simple model (usually a decision tree) to make initial predictions.

2. **Step 2: Compute Residuals**

   Compute the residuals (errors) of the model by subtracting the predicted values from the true values.

3. **Step 3: Train a New Model**

   Train a new model to predict the residuals.

4. **Step 4: Update the Model**

   Add the predictions from the new model to the existing model.

5. **Step 5: Repeat**

   Repeat this process for a set number of iterations (i.e., estimators). Each iteration tries to correct the errors of the previous model.

This completes the explanations for all 7 assignments. Let me know if you need any further clarifications or adjustments!