Gradient Descent

```
import numpy as np
import pandas as pd

# Load data
data = pd.read_csv('datasets/advertising/Advertising Budget and Sales.csv')
print(data.head())

# Preprocessing
print(data.isnull().sum())
data_cleaned = data.drop(['Unnamed: 0'], axis=1)
print("Cleaned data")
print(data_cleaned.isnull().sum())

# Select Radio ad budget as feature and sales as target
X = data_cleaned[['Radio Ad Budget ($)']]
y = data_cleaned[['Sales ($)']]

X = np.array(X)
y = np.array(y).reshape(-1,1)

# Normalize the features for stability
X = (X - np.min(X)) / (np.max(X) - np.min(X))

# Hyperparameters
learning_rate = 0.001
epochs = 1000
weights = np.random.rand(2,1)

print("Initial weights", weights)

# Gradient Descent
for epoch in range(epochs):
  bias = np.ones((len(X), 1))
  X_b = np.c_[bias, X]  # Add bias term to X
  y_pred = X_b.dot(weights)  # Predictions
  error = y_pred - y  # Error
  gradients = 2 * X_b.T.dot(error) / len(X_b)  # Gradients for weights
  weights -= learning_rate * gradients  # Update weights

  mse = np.mean(error ** 2)  # Mean Squared Error
  if epoch % 100 == 0:
    print(f"Epoch {epoch}, MSE: {mse}")

# Print final weights
print("Final weights", weights)

# Evaluate model using RMSE
rmse = np.sqrt(np.mean(error ** 2))
print(f"RMSE: {rmse}")
```

- Gradient Descent is an optimization algorithm used to minimize the error in machine learning models, typically by adjusting model parameters (like weights in linear regression)
- The goal is to learn a model (like $y=wx+b$) that can accurately predict y given X
- The cost function, or the loss function, quantifies the error between the predicted values $y$pred and the actual values y. For linear regression, this is typically the MSE
- The gradient represents how much the cost function changes to the weights and bias.
- Positive gradient: If the slope is positive, increasing the weight would increase the error.
- Negative gradient: If the slope is negative, increasing the weight would decrease the error.

SVM (All types included)

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix,accuracy_score, recall_score, precision_score,
f1_score

file_path = "datasets/Social network dataset/Social_Network_Ads.csv"
data = pd.read_csv(file_path)
print("Initial data")
print(data.head())

#preprocessing
print(data.isnull().sum())

label_encoder = LabelEncoder()
data['Gender'] = label_encoder.fit_transform(data['Gender'])
X = data[['Gender','Age','EstimatedSalary']]
y = data['Purchased']

X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=42)

model = SVC(kernel='linear')
# model = SVC(kernel='rbf')
model.fit(X_train,y_train)

y_pred = model.predict(X_test)
cm = confusion_matrix(y_test,y_pred)
acc_s = accuracy_score(y_test,y_pred)
recall_s = recall_score(y_test,y_pred)
precision_s = precision_score(y_test,y_pred)
f1_s = f1_score(y_test,y_pred)
print("Confusion matrix")
print(cm)
print(f"Accuracy: {acc_s}")
print(f"Recall: {recall_s}")
print(f"Precision: {precision_s}")
print(f"F1 Score: {f1_s}")
print("Predictions")
for actual,predicted in zip(y_test,y_pred):
    print(f"Actual: {actual}, Predicted: {predicted}")
```

- Support Vector Machine (SVM) is a supervised learning algorithm for classification and regression tasks. Its main goal is to find a hyperplane that best separates the data points of different classes.
- It tries to maximize the margin (distance) between the data points from different classes and the hyperplane.
- Two types - linear and non-linear(rbf - radial basis function: maps it into higher dimensions where data becomes linearly separable)
- LabelEncoder is used to convert categorical data (e.g., text labels like "Male", "Female") into numerical data.
- A Confusion Matrix is a table that helps evaluate the performance of a classification model. It shows the count of True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN).
- Accuracy = $\frac{TP+TN}{TP+TN+FP+FN}$
- Recall = $\frac{TP}{TP + FN}$
- Precision = $\frac{TP}{TP + FP}$
- F1 = 2 x $\frac{Precision\ x\ Recall}{Precision + Recall}$

Decision Tree (All types)

```
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

file_path = "datasets/Iris/Iris.csv"
data = pd.read_csv(file_path)
print("Initial data")
print(data.head())

print(data.isnull().sum())
X = data[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']]
y = data['Species']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model = DecisionTreeClassifier(criterion='gini',max_depth=4,min_samples_split=2)
# model = DecisionTreeClassifier(criterion='entropy',max_depth=4,min_samples_split=2)
# model = DecisionTreeClassifier(criterion='log_loss',max_depth=4,min_samples_split=2)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
acc_s = accuracy_score(y_test, y_pred)
print(f"Accuracy: {acc_s}")
```

```
example_features = [[5.1, 3.5, 1.4, 0.2]]
predictions = model.predict(example_features)
print(f"Category for {example_features[0]} is {predictions[0]}")
```

- A Decision Tree is a supervised learning algorithm for classification and regression tasks.
- It breaks down the dataset into smaller subsets based on feature values, creating a tree-like structure of decisions.
- Each internal node in the tree represents a "test" on a feature, each branch represents the outcome of the test, and each leaf node represents the final decision or classification.
- Gini Impurity: Measures impurity by calculating the probability of misclassification. Faster to compute.
- Entropy: Measures the amount of information needed to classify data, aiming to reduce uncertainty (impurity).
- Log Loss: Measures the accuracy of probability predictions; not commonly used for decision trees but more for probabilistic models.

CLASSIFIERS (All types)

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier,
BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

file_path = "datasets/Iris/Iris.csv"
data = pd.read_csv(file_path)
print("Initial data")
print(data.head())

print(data.isnull().sum())
X = data[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']]
y = data['Species']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model =
BaggingClassifier(base_estimator=DecisionTreeClassifier(),n_estimators=50,random_state=42)
# model = RandomForestClassifier(n_estimators=50,random_state=42)
# model = GradientBoostingClassifier(n_estimators=50,random_state=42)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
accuracy_score = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy_score}")
print("Predictions")
for actual, predicted in zip(y_test, y_pred):
    print(f"Actual: {actual}, Predicted: {predicted}")
```

- RandomForestClassifier - An ensemble method that builds multiple decision trees using different random subsets of the data and features. The final prediction is made by averaging or majority voting from all trees.
- Advantage: Reduces overfitting, robust, and works well on a wide variety of datasets.
- Disadvantage: Can be less interpretable and slower for making predictions due to multiple trees.
- GradientBoostingClassifier - A sequential ensemble method where each new decision tree tries to correct the errors made by the previous trees, using gradient descent to minimize the loss function.
- Advantage: High accuracy, especially on complex datasets, and customizable loss functions.
- Disadvantage: Prone to overfitting and can be slow to train due to sequential learning.
- BaggingClassifier - An ensemble method that trains multiple base models (e.g., decision trees) on random subsets of the data, aggregating their predictions to improve stability and accuracy.
- Advantage: Reduces variance and prevents overfitting.
- Disadvantage: Does not significantly reduce bias, and may not perform well on noisy or imbalanced datasets.
- AdaBoostClassifier: A boosting algorithm that builds multiple weak learners (often decision stumps, i.e., single-level decision trees) sequentially. Each new learner focuses on correcting the mistakes of the previous ones by adjusting the weights of the incorrectly classified data points.
- Advantage: Improves accuracy by focusing on hard-to-classify examples, works well with weak learners, and reduces bias.
- Disadvantage: Sensitive to noisy data and outliers, and can overfit on complex datasets if not carefully tuned.

Principal Component Analysis

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.decomposition import PCA

file_path = "datasets/Iris/Iris.csv"
data = pd.read_csv(file_path)
print("Initial data")
print(data.head())

print(data.isnull().sum())
X = data[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']]
y = data['Species']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

dt_classifier = DecisionTreeClassifier(max_depth=4,random_state=42)
dt_classifier.fit(X_train, y_train)
y_pred_without_pca = dt_classifier.predict(X_test)

accuracy_score_without_pca = accuracy_score(y_test, y_pred_without_pca)
```

```
print(f"Accuracy without PCA: {accuracy_score_without_pca}")

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(X_pca, y, test_size=0.2,
random_state=42)

dt_classifier_pca = DecisionTreeClassifier(max_depth=4,random_state=42)
dt_classifier_pca.fit(X_train_pca, y_train_pca)
y_pred_with_pca = dt_classifier_pca.predict(X_test_pca)

accuracy_score_with_pca = accuracy_score(y_test_pca, y_pred_with_pca)
print(f"Accuracy with PCA: {accuracy_score_with_pca}")

print("Comparing the accuracies")
print(f"Accuracy without PCA: {accuracy_score_without_pca}")
print(f"Accuracy with PCA: {accuracy_score_with_pca}")
```

- PCA (Principal Component Analysis) is a dimensionality reduction technique that transforms data into a lower-dimensional space while preserving as much variance (information) as possible. It helps simplify complex datasets, reduce computation time, and prevent overfitting, especially when dealing with large feature sets.

Logistic Regression

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score, precision_score

class LogisticRegression:
    def __init__(self,learning_rate=0.01,epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = None
        self.bias = None

    def sigmoid(self,z):
        return 1/(1+np.exp(-z))

    def fit(self,X,y):
        num_samples,num_features = X.shape
        self.weights = np.zeros(num_features)
        self.bias = 0
        for _ in range(self.epochs):
            linear_model = np.dot(X,self.weights) + self.bias
            predictions = self.sigmoid(linear_model)
            dw = (1/num_samples)*np.dot(X.T,(predictions-y))
            db = (1/num_samples)*np.sum(predictions-y)
            self.weights -= self.learning_rate*dw
            self.bias -= self.learning_rate*db

    def predict(self,X):
        linear_model = np.dot(X,self.weights) + self.bias
        predictions = self.sigmoid(linear_model)
        predictions_labels = [1 if i>0.5 else 0 for i in predictions]
        return predictions_labels

class OVRLogisticRegression:
    def __init__(self, lr=0.01, epochs=1000):
        self.lr = lr
        self.epochs = epochs
        self.classifiers = []

    def fit(self, X, y):
        self.classes = np.unique(y)
        for c in self.classes:
            y_binary = np.where(y == c, 1, 0)
            clf = LogisticRegression(learning_rate= self.lr, epochs = self.epochs)
            clf.fit(X, y_binary)
            self.classifiers.append(clf)

    def predict(self, X):
        class_scores = []
        for clf in self.classifiers:
            class_scores.append(clf.predict(X))

        class_scores = np.array(class_scores).T
        return np.argmax(class_scores, axis = 1)

df = pd.read_csv("datasets/Iris/Iris.csv")
print(df.head())

label_encoder = LabelEncoder()
df['Species'] = label_encoder.fit_transform(df['Species'])

X = df.iloc[:, 1:5] # All 4 features
y = df['Species']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
ovr_log_reg = OVRLogisticRegression()

ovr_log_reg.fit(X_train, y_train)
y_pred = ovr_log_reg.predict(X_test)

conf_matrix = confusion_matrix(y_test, y_pred)
```

```
accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred, average='micro')
precision = precision_score(y_test, y_pred, average='micro')

print(f"CONFUSION MATRIX:\n {conf_matrix}")
print(f"ACCURACY:  {accuracy * 100:.2f}%")
print(f"RECALL:  {recall * 100:.2f}%")
print(f"Precision: {precision*100:.2f}%")
```

- Logistic regression is a classification algorithm used to predict binary outcomes. It models the probability that a given input belongs to a particular class by applying the sigmoid function to a linear combination of the input features and weights.
- The sigmoid function is the core of logistic regression, mapping the model's linear output to a probability between 0 and 1: $\frac{1}{1+e^{-z}}$
- For multi-class classification, logistic regression can be extended using the One-vs-Rest (OvR) approach. In OvR, a separate binary classifier is trained for each class, distinguishing one class from all others. At prediction time, each classifier outputs a score, and the class with the highest score is selected as the final prediction.