Introduction to Machine Learning (DIT-S563)

LAB EXPERIMENTS

1. Exploratory Data Analysis (EDA):

1. Environment setup, Python, and library installation.

Environment Setup:

1. Before you begin, make sure you have Python installed on your system. You can download Python from the official website (https://www.python.org/downloads/).
2. Optionally, it's recommended to set up a virtual environment to manage dependencies. To create a virtual environment, use the following commands in your terminal:

```
# Install virtualenv (if not already installed)
pip install virtualenv

# Create a virtual environment
virtualenv myenv

# Activate the virtual environment
# On Windows, use: myenv\Scripts\activate

source myenv/bin/activate
```

Library Installation:

3. Once you have Python and your virtual environment set up (if used), you can install the required libraries. For this lab, you'll need Matplotlib and Pandas. Install them using pip:

```
pip install matplotlib pandas
```

Verify that the libraries are successfully installed by importing them in a Python script or interactive Python environment:

```
import matplotlib
import pandas

print ("Matplotlib version:", matplotlib.__version__)
print ("Pandas version:", pandas.__version__)
```

2. Data visualization and summary statistics using libraries like Matplotlib and Pandas.

Now that you have Python and the required libraries installed, you can perform data visualization and summary statistics tasks. Here's an example:

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
# Sample data (you can replace this with your dataset)
data = {'Age': [25, 30, 35, 40, 45],
        'Salary': [50000, 60000, 75000, 90000, 100000]}
# Create a Pandas DataFrame from the sample data
df = pd.DataFrame(data)
# Data Visualization: Create a scatter plot
plt.scatter(df['Age'], df['Salary'])
plt.title('Scatter Plot of Age vs. Salary')
plt.xlabel('Age')
plt.ylabel('Salary')
plt.show()
# Summary Statistics
summary = df.describe()
print(summary)
```

In this code:

We import Pandas for data manipulation and Matplotlib for data visualization.

We create a simple dataset in the form of a dictionary and convert it into a Pandas DataFrame.

We create a scatter plot to visualize the relationship between Age and Salary.

We use describe () to generate summary statistics for the dataset.

Make sure to replace the sample data with your actual dataset when working on real projects.

2. Data Preprocessing:

1. Handling missing data (imputation).

To handle missing data in a CSV file using Python and then save the updated data in a new CSV file, you can use the Pandas library. Here's a Python code example to demonstrate this:
Assuming you have a CSV file named "input_data.csv" with missing data, and you want to save the cleaned data in "output_data.csv," here's the code:

```
import pandas as pd
# Load the CSV file with missing data
input_csv_file = 'input_data.csv'
df = pd.read_csv(input_csv_file)
# Handling missing data (e.g., replace missing values with the mean of the column)
df.fillna(df.mean(), inplace=True)
# Save the cleaned data to a new CSV file
output_csv_file = 'output_data.csv'
df.to_csv(output_csv_file, index=False)
print ("Missing data handling and cleaned data saved to", output_csv_file)
```

In this code:

1. We import the Pandas library.

2. We load the CSV file "input_data.csv" using `pd.read_csv()` to create a Pandas DataFrame.

3. We use `fillna()` to replace missing values with the mean of the respective columns. You can replace the filling strategy with other methods as needed, such as using a specific value or forward/backward filling.

4. We save the cleaned data to a new CSV file named "output_data.csv" using `to_csv()`. The `index=False` argument prevents writing the DataFrame index to the CSV file.

5. Finally, we print a message indicating that the missing data has been handled and the cleaned data has been saved.

Ensure that you replace "input_data.csv" with the actual filename and path of your input CSV file and specify the desired name for the output CSV file. This code snippet will read the input data, handle missing values, and save the cleaned data to the specified output CSV file.

2. Data encoding (categorical to numerical).

To encode categorical data into numerical format, you can use techniques like one-hot encoding or label encoding, depending on the nature of your data. Here, I'll provide examples for both methods.

Method 1: Label Encoding

Label encoding assigns a unique integer to each category in a categorical feature. Here's a Python code example using the `LabelEncoder` from the scikit-learn library:

```
from sklearn.preprocessing import LabelEncoder
# Sample data
data = {'Category': ['Red', 'Green', 'Blue', 'Red', 'Green']}
# Create a Pandas DataFrame from the sample data
df = pd.DataFrame(data)
# Initialize the label encoder
label_encoder = LabelEncoder()
# Apply label encoding to the 'Category' column
df['Category_encoded'] = label_encoder.fit_transform(df['Category'])
# Display the encoded DataFrame
print(df)
```

In this example, the 'Category' column is encoded into numerical values. The `fit_transform` method of the label encoder is used to transform the categorical data.

Method 2: One-Hot Encoding

One-hot encoding converts categorical variables into binary vectors, where each category is represented as a binary column. Here's a Python code example using the `get_dummies` function from Pandas:

```
# Sample data
data = {'Category': ['Red', 'Green', 'Blue', 'Red', 'Green']}

# Create a Pandas DataFrame from the sample data
df = pd.DataFrame(data)

# Perform one-hot encoding using Pandas
df_encoded = pd.get_dummies(df, columns=['Category'], prefix=['Category'])

# Display the one-hot encoded DataFrame
print(df_encoded)
```

In this example, the 'Category' column is one-hot encoded into separate binary columns for each category. Choose the encoding method (label encoding or one-hot encoding) based on your data and the requirements of your machine learning model. Label encoding is suitable for ordinal categorical data, while one-hot encoding is often used for nominal categorical data.

3. Feature scaling and normalization.

Certainly, feature scaling and normalization are important preprocessing steps in many machine learning tasks. Here's Python code for both Min-Max scaling and Z-score (Standardization) normalization using the `scikit-learn` library:

```
# Import necessary libraries
import pandas as pd
from sklearn.preprocessing import MinMaxScaler, StandardScaler

# Sample data (you can replace this with your dataset)
data = {'Age': [25, 30, 35, 40, 45],
        'Salary': [50000, 60000, 75000, 90000, 100000]}

# Create a Pandas DataFrame from the sample data
df = pd.DataFrame(data)

# Min-Max Scaling
min_max_scaler = MinMaxScaler()
scaled_data = min_max_scaler.fit_transform(df)
df_min_max_scaled = pd.DataFrame(scaled_data, columns=df.columns)

print("Min-Max Scaled Data:")
print(df_min_max_scaled)

# Z-score (Standardization) Normalization
standard_scaler = StandardScaler()
normalized_data = standard_scaler.fit_transform(df)
df_standardized = pd.DataFrame(normalized_data, columns=df.columns)

print("\nStandardized Data (Z-score Normalization):")
print(df_standardized)
```

In this code:
1. We import Pandas for data manipulation and the required functions from `sklearn.preprocessing` to perform Min-Max scaling and Standardization.
2. We create a simple dataset (replace with your actual dataset) and convert it into a Pandas DataFrame.
3. Min-Max Scaling is applied using `MinMaxScaler`, which scales features to a specified range (typically between 0 and 1).
4. Z-score Normalization (Standardization) is applied using `StandardScaler`, which scales features to have a mean of 0 and a standard deviation of 1.
Both `df_min_max_scaled` and `df_standardized` will contain the scaled/normalized data, and you can use them for further machine learning tasks.

3. Supervised Learning:

1. Linear regression for a simple predictive task.

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# Generate sample data (you can replace this with your dataset)
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)  # Feature (independent variable)
y = np.array([2, 4, 5, 4, 5])  # Target (dependent variable)

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a linear regression model
model = LinearRegression()

# Fit the model to the training data
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Plot the data and the regression line
plt.scatter(X, y, label='Actual Data', color='blue')
plt.plot(X_test, y_pred, label='Regression Line', color='red')
plt.xlabel('Feature (X)')
plt.ylabel('Target (y)')
plt.legend()
plt.show()

# Print the model parameters
print("Intercept (b0):", model.intercept_)
print("Coefficient (b1):", model.coef_[0])

# Calculate the coefficient of determination (R-squared)
r_squared = model.score(X_test, y_test)
print("R-squared:", r_squared)
```

In this code:
We import the necessary libraries, including numpy for numerical operations, matplotlib for data visualization, and scikit-learn for machine learning.
We generate sample data (X and y), but you should replace this with your own dataset.
We split the data into training and testing sets to evaluate the model's performance.

We create a Linear Regression model, fit it to the training data, and make predictions on the test data.
We plot the data points and the regression line.
We print the model parameters, including the intercept and coefficient.
We calculate the coefficient of determination (R-squared) to evaluate the model's goodness of fit.

2. Logistic regression for binary classification.

We'll use scikit-learn to create a logistic regression model for binary classification.
First, make sure you have scikit-learn installed. If not, you can install it using `pip`:

pip install scikit-learn

Now, let's write the code:

```
# Import necessary libraries
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Sample data for binary classification (replace with your dataset)
# X contains the features, and y contains the target labels (0 or 1)
X = np.array([[1.2, 2.3], [2.8, 3.7], [4.1, 5.2], [6.2, 6.8], [7.5, 8.9]])
y = np.array([0, 0, 1, 1, 1])

# 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, random_state=42)

# Create a Logistic Regression model
logistic_reg_model = LogisticRegression()

# Train the model on the training data
logistic_reg_model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = logistic_reg_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print("Model Accuracy:", accuracy)
print("Confusion Matrix:")
print(conf_matrix)
print("Classification Report:")
print(class_report)
```

In this code:
- We import necessary libraries, including NumPy for data handling, scikit-learn for building the logistic regression model, and metrics for evaluation.
- We create a sample dataset (`X` and `y`) for binary classification. You should replace this with your own dataset.
- We split the dataset into training and testing sets using `train_test_split`.
- We create a Logistic Regression model, fit it to the training data, and make predictions on the test data.
- Finally, we evaluate the model's performance using accuracy, a confusion matrix, and a classification report.

Make sure to replace the sample data with your actual dataset when working on real classification tasks.

3. k-Nearest Neighbors (k-NN) for classification.

Python code example for implementing k-Nearest Neighbors (k-NN) for classification using the popular scikit-learn library. In this example, we'll use a sample dataset for classification:

```
# Import necessary libraries
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Load a sample dataset (Iris dataset)
iris = datasets.load_iris()
X = iris.data  # Features
y = iris.target  # Target (classes)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create a k-NN classifier with k=3
k = 3
knn_classifier = KNeighborsClassifier(n_neighbors=k)

# Fit the classifier to the training data
knn_classifier.fit(X_train, y_train)

# Make predictions on the test data
y_pred = knn_classifier.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

In this code:
- We import the necessary libraries, including scikit-learn, which provides tools for machine learning tasks.

- We load a sample dataset (Iris dataset) using scikit-learn's built-in datasets.
- The data is split into training and testing sets using `train_test_split`.
- We create a k-NN classifier with `KNeighborsClassifier` and set `n_neighbors` to the desired value of k.
- The classifier is trained on the training data using the `fit` method.
- We make predictions on the test data using the `predict` method.
- Finally, we calculate the accuracy of the model by comparing the predicted labels to the actual labels in the test set.

You can replace the Iris dataset with your own dataset and adjust the value of `k` as needed for your classification task.

4. Decision Trees for classification and regression.

Python code example for using a Decision Tree for classification using the popular scikit-learn library. This code demonstrates the process of training a Decision Tree classifier on a dataset and making predictions.

```
# Import the necessary libraries
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load a sample dataset (Iris dataset for classification)
data = datasets.load_iris()
X = data.data  # Features
y = data.target  # Target (labels)

# Split the dataset into a training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create a Decision Tree classifier
clf = DecisionTreeClassifier()

# Train (fit) the classifier on the training data
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)

# Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

# Print the results
print("Accuracy:", accuracy)
print("\nConfusion Matrix:\n", conf_matrix)
```

```
print("\nClassification Report:\n", classification_rep)
```

In this code:

1. We import the necessary libraries, including scikit-learn for the Decision Tree classifier.
2. We load a sample dataset (Iris dataset) for classification. You can replace this with your own dataset.
3. We split the dataset into training and testing sets using `train_test_split`.
4. We create a Decision Tree classifier using `DecisionTreeClassifier`.
5. We train the classifier on the training data using the `fit` method.
6. We make predictions on the test data using the `predict` method.
7. We evaluate the model's performance by calculating accuracy, creating a confusion matrix, and generating a classification report.
8. Finally, we print the results, including accuracy, the confusion matrix, and the classification report.

You can replace the Iris dataset with your own dataset for classification by loading your data into `X` (features) and `y` (target labels).

5. Naive Bayes for text classification

Python code example for text classification using the Naive Bayes algorithm. In this example, we'll use the `sklearn` library, which provides tools for machine learning and natural language processing tasks. We'll use the Multinomial Naive Bayes classifier for text classification, which is commonly used for tasks like spam detection and sentiment analysis.

let's create a simple text classification example:

```
# Import necessary libraries
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# Sample text data and corresponding labels
texts = ["This is a positive message.", "Negative feedback.", "Positive sentiment.", "Not a good sign.", "Great experience."]
labels = ["positive", "negative", "positive", "negative", "positive"]

# Initialize the CountVectorizer to convert text data into numerical features
vectorizer = CountVectorizer()

# Convert the text data into a document-term matrix
X = vectorizer.fit_transform(texts)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2, random_state=42)

# Initialize the Multinomial Naive Bayes classifier
```

```
clf = MultinomialNB()

# Train the classifier on the training data
clf.fit(X_train, y_train)

# Predict the labels for the test data
y_pred = clf.predict(X_test)

# Calculate and print the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Display a classification report with precision, recall, and F1-score
report = classification_report(y_test, y_pred)
print("Classification Report:\n", report)
```

In this code:

1. We import the necessary libraries, including `CountVectorizer` for converting text data into numerical features, `MultinomialNB` for the Naive Bayes classifier, and other tools for evaluation.
2. We provide a list of sample text data and corresponding labels. You should replace this with your own dataset.
3. We use `CountVectorizer` to convert the text data into a document-term matrix.
4. The data is split into training and testing sets using `train_test_split`.
5. We initialize the Multinomial Naive Bayes classifier, train it on the training data, and make predictions on the test data.
6. We calculate and print the accuracy of the classifier, which tells us how well the model performs.
7. We also display a classification report that provides additional evaluation metrics, such as precision, recall, and F1-score.

Remember to replace the sample data with your own dataset for real-world text classification tasks.

6. Support Vector Machines (SVM) for classification

Python code for using Support Vector Machines (SVM) for classification using the popular scikit-learn library. In this example, we'll use the Iris dataset for a simple classification task:

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report

# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data
```

```
y = iris.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create an SVM classifier
clf = SVC(kernel='linear', C=1)  # You can choose different kernels (e.g., 'linear', 'rbf') and
adjust the regularization parameter C

# Train the SVM classifier on the training data
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Generate a classification report
report = classification_report(y_test, y_pred, target_names=iris.target_names)
print(report)

# Visualize the decision boundary (for two features only)
# Note: This visualization is for illustrative purposes and works only with two features
if X.shape[1] == 2:
   plt.figure(figsize=(8, 6))
   plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)

   # plot the decision function
   ax = plt.gca()
   xlim = ax.get_xlim()
   ylim = ax.get_ylim()

   xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 50),
                        np.linspace(ylim[0], ylim[1], 50))
   Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
   Z = Z.reshape(xx.shape)

   plt.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
   plt.title('SVM Decision Boundary')
   plt.xlabel(iris.feature_names[0])
   plt.ylabel(iris.feature_names[1])
   plt.show()
```

In this code:
- We load the Iris dataset and split it into training and testing sets.

- We create an SVM classifier using scikit-learn's `SVC` class. You can specify different kernels and adjust the regularization parameter `C` according to your problem.
- We train the SVM classifier on the training data and make predictions on the test data.
- We calculate the accuracy of the model and generate a classification report to assess its performance.
- If the dataset has two features (as in Iris), we visualize the decision boundary. Note that this visualization works only for 2D data.

Make sure to adapt the code to your specific dataset and problem. You can also explore different SVM kernels and hyperparameters to improve the model's performance.

4. Model Evaluation:

1. Evaluation metrics: accuracy, precision, recall, F1-score, ROC curves, etc.

Python code snippet that demonstrates how to calculate and evaluate the following metrics: accuracy, precision, recall, F1-score, and ROC curves. Note that this code assumes you have predicted values and true labels, typically from a classification task.

```
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_curve, auc
import matplotlib.pyplot as plt

# Sample true labels and predicted values (replace with your data)
true_labels = np.array([1, 0, 1, 1, 0, 1, 0, 0, 1, 1])
predicted_values = np.array([1, 0, 1, 0, 1, 1, 0, 1, 1, 0])

# Accuracy
accuracy = accuracy_score(true_labels, predicted_values)
print("Accuracy:", accuracy)

# Precision
precision = precision_score(true_labels, predicted_values)
print("Precision:", precision)

# Recall
recall = recall_score(true_labels, predicted_values)
print("Recall:", recall)

# F1-Score
f1 = f1_score(true_labels, predicted_values)
print("F1-Score:", f1)

# ROC Curve
fpr, tpr, _ = roc_curve(true_labels, predicted_values)
roc_auc = auc(fpr, tpr)

# Plot ROC Curve
```

```
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

In this code:

- We import the necessary modules from scikit-learn (`sklearn.metrics`) for the metrics and `matplotlib` for plotting.
- We provide sample true labels and predicted values. Replace these arrays with your actual data.
- We calculate accuracy, precision, recall, and F1-score using scikit-learn's functions.
- We calculate the ROC curve by obtaining false positive rates (`fpr`) and true positive rates (`tpr`) and calculate the area under the ROC curve (`roc_auc`).
- Finally, we plot the ROC curve to visualize the model's performance in binary classification tasks.

Make sure to adjust the true labels and predicted values with your own data for meaningful evaluation.

2. Cross-validation to assess model performance.

Cross-validation is a crucial step to assess the performance of a machine learning model. Here's Python code that demonstrates how to perform cross-validation using the `cross_val_score` function from the scikit-learn library:

```
# Import necessary libraries
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.datasets import load_iris  # Example dataset
from sklearn.tree import DecisionTreeClassifier  # Example model

# Load a sample dataset (you should replace this with your dataset)
data = load_iris()
X, y = data.data, data.target

# Create a machine learning model (you should replace this with your model)
model = DecisionTreeClassifier()

# Define the number of folds for cross-validation
num_folds = 5  # You can adjust this as needed

# Create a cross-validation object (in this case, using K-Fold cross-validation)
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)
```

```
# Perform cross-validation and get the accuracy scores
scores = cross_val_score(model, X, y, cv=kf)

# Print the cross-validation scores
print("Cross-validation scores:", scores)

# Calculate and print the mean and standard deviation of the scores
mean_score = scores.mean()
std_score = scores.std()
print("Mean accuracy:", mean_score)
print("Standard deviation of accuracy:", std_score)
```

In this code:
- We import the necessary libraries from scikit-learn.
- We load a sample dataset (Iris dataset) for demonstration purposes. You should replace it with your dataset.
- We create a machine learning model (a Decision Tree Classifier in this example).
- We define the number of folds for cross-validation, which determines how many times the data will be split and tested.
- We create a cross-validation object, in this case using K-Fold cross-validation with the specified number of folds.
- We perform cross-validation using `cross_val_score`, which returns an array of accuracy scores for each fold.
- Finally, we calculate and print the mean accuracy and standard deviation of the accuracy scores to assess the model's performance.

Adjust the dataset, model, and number of folds according to your specific use case and dataset.

5. Hyperparameter Tuning:

1. Hyperparameter optimization

Hyperparameter optimization is a crucial step in machine learning to fine-tune your model's performance. You can use libraries like scikit-learn and GridSearchCV or RandomizedSearchCV to perform hyperparameter optimization. Below is a Python code example using scikit-learn and GridSearchCV for hyperparameter optimization with a Support Vector Machine (SVM) classifier:

```
# Import necessary libraries
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# Load a sample dataset (Iris dataset in this example)
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create an SVM classifier
svm = SVC()

# Define the hyperparameters and their possible values to search
param_grid = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'rbf', 'poly'],
    'gamma': [0.001, 0.01, 0.1, 'scale', 'auto']
}

# Create a GridSearchCV object
grid_search = GridSearchCV(svm, param_grid, cv=5, n_jobs=-1, verbose=2)

# Fit the grid search to the data to find the best hyperparameters
grid_search.fit(X_train, y_train)

# Get the best hyperparameters and the best estimator
best_params = grid_search.best_params_
best_estimator = grid_search.best_estimator_

# Print the best hyperparameters
print("Best Hyperparameters:", best_params)

# Evaluate the best model on the test set
accuracy = best_estimator.score(X_test, y_test)
print("Test Set Accuracy:", accuracy)
```

In this code:
- We load the Iris dataset from scikit-learn as an example.
- We split the dataset into training and testing sets.
- We create an SVM classifier.
- We define a dictionary of hyperparameters and their possible values to search over.
- We create a GridSearchCV object with the SVM classifier and the hyperparameter grid.
- We fit the grid search to the training data to find the best hyperparameters.
- We print the best hyperparameters and evaluate the best model on the test set.

You can adapt this code to your specific dataset and machine learning model by changing the dataset loading and the classifier used.

2. Fine-tuning model parameters for improved performance.

Fine-tuning model parameters is an essential step in optimizing your machine learning models for improved performance. In this example, we'll use a simple dataset and a scikit-learn classifier to demonstrate how to fine-tune model parameters using grid search. Grid search helps you search through different hyperparameter combinations to find the best set of parameters for your model.

Here's a Python code example for fine-tuning a model using grid search:

```
# Import necessary libraries
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestClassifier

# Load a sample dataset (Iris dataset)
data = load_iris()
X = data.data
y = data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a random forest classifier
classifier = RandomForestClassifier(random_state=42)

# Define hyperparameters and their possible values for grid search
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Perform grid search with cross-validation to find the best parameters
grid_search = GridSearchCV(classifier, param_grid, cv=5, n_jobs=-1)  # 'cv' is the number of cross-validation folds
grid_search.fit(X_train, y_train)

# Print the best hyperparameters
best_params = grid_search.best_params_
print("Best Hyperparameters:")
for param, value in best_params.items():
    print(f"{param}: {value}")

# Evaluate the model with the best parameters on the test set
best_model = grid_search.best_estimator_
```

```
accuracy = best_model.score(X_test, y_test)
print("Test Set Accuracy:", accuracy)
```

In this code:
- We use the Iris dataset as an example.
- We split the dataset into training and testing sets.
- We create a RandomForestClassifier from scikit-learn.
- We define a grid of hyperparameters and their possible values in the `param_grid` dictionary.
- We perform grid search using `GridSearchCV` to find the best combination of hyperparameters.
- We print the best hyperparameters and evaluate the model's performance on the test set using the best parameters.

Make sure to replace the dataset and classifier with your specific dataset and machine learning model. Grid search is a versatile technique that can be applied to various machine learning models and datasets to fine-tune hyperparameters for improved performance.

6. Ensemble Learning:

1. Random Forests for classification and regression.

Random Forests can be used for both classification and regression tasks. Here's an example of Python code for both classification and regression using Random Forests:

Random Forest for Classification:

```
# Import necessary libraries
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report

# Sample data for classification (replace with your dataset)
X, y = your_features, your_labels

# 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, random_state=42)

# Create a Random Forest Classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the model to the training data
rf_classifier.fit(X_train, y_train)

# Make predictions on the test data
y_pred = rf_classifier.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
```

```
print("Accuracy:", accuracy)

# Generate a classification report
report = classification_report(y_test, y_pred)
print("Classification Report:\n", report)
```

In this code:
- We import the necessary libraries, including RandomForestClassifier from scikit-learn.
- Replace `your_features` and `your_labels` with your actual data.
- We split the data into training and testing sets.
- We create a Random Forest Classifier, fit it to the training data, and make predictions on the test data.
- We evaluate the model using accuracy and a classification report.

Random Forest for Regression:

```
# Import necessary libraries
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Sample data for regression (replace with your dataset)
X, y = your_features, your_target

# 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, random_state=42)

# Create a Random Forest Regressor
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)

# Fit the model to the training data
rf_regressor.fit(X_train, y_train)

# Make predictions on the test data
y_pred = rf_regressor.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print("Mean Squared Error:", mse)
print("R-squared:", r2)
```

In this code:
- We import the necessary libraries, including RandomForestRegressor from scikit-learn.
- Replace `your_features` and `your_target` with your actual data.
- We split the data into training and testing sets.
- We create a Random Forest Regressor, fit it to the training data, and make predictions on the test data.
- We evaluate the model using mean squared error (MSE) and R-squared.

2. Gradient Boosting (e.g., XGBoost) for improved predictive performance.

```
# Import necessary libraries
import numpy as np
import pandas as pd
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load your dataset (replace 'your_dataset.csv' with your dataset file)
data = pd.read_csv('your_dataset.csv')

# Assuming your dataset has a 'target' column for the labels
X = data.drop('target', axis=1)
y = data['target']

# 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, random_state=42)

# Initialize the XGBoost classifier
model = XGBClassifier()

# Train the model on the training data
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Display a classification report and confusion matrix
print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

In this code:
You load your dataset using Pandas. Make sure to replace 'your_dataset.csv' with the actual file path to your dataset.
You split the data into training and testing sets using train_test_split from scikit-learn.
An XGBoost classifier is initialized, trained on the training data, and used to make predictions on the test data.
Accuracy is calculated using accuracy_score from scikit-learn.
A classification report and confusion matrix are displayed to evaluate the model's performance.
Be sure to replace the dataset file and column names according to your specific dataset. XGBoost offers various hyperparameters that you can tune for better performance, but this code provides a basic example to get you started.

7. Unsupervised Learning:

1. K-Means clustering for data grouping.

Python code for K-Means clustering using the popular `scikit-learn` library. In this example, we'll generate some random data and then apply K-Means clustering to it.

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Generate random data (you can replace this with your dataset)
n_samples = 300
n_features = 2
n_clusters = 3

data, labels = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_clusters,
random_state=42)

# Create a K-Means model with the desired number of clusters
kmeans = KMeans(n_clusters=n_clusters)

# Fit the K-Means model to the data
kmeans.fit(data)

# Get cluster assignments for each data point
cluster_labels = kmeans.labels_

# Get the coordinates of the cluster centers
cluster_centers = kmeans.cluster_centers_

# Visualize the clusters
plt.figure(figsize=(8, 6))
plt.scatter(data[:, 0], data[:, 1], c=cluster_labels, cmap='viridis', s=50)
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1, c='red', marker='x', s=200)
plt.title('K-Means Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

In this code:

- We import the necessary libraries, including NumPy, Matplotlib, and scikit-learn.
- We generate random data using `make_blobs()`, but you can replace it with your dataset.
- We create a K-Means model with a specified number of clusters using `KMeans()` from scikit-learn.

- We fit the K-Means model to the data using the `fit()` method.
- We obtain cluster assignments for each data point using `kmeans.labels_`.
- We get the coordinates of the cluster centers using `kmeans.cluster_centers_`.
- We visualize the clusters by creating a scatter plot, where data points are colored by their cluster assignments, and cluster centers are marked with red 'x' symbols.

8. Dimensionality Reduction:

1. Feature extraction techniques (e.g., PCA).

Python code example for Principal Component Analysis (PCA) using the popular machine learning library, scikit-learn. PCA is used for dimensionality reduction, which is particularly useful when dealing with high-dimensional datasets.

```
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Create a sample dataset
np.random.seed(0)
data = np.random.randn(100, 2)  # 100 samples, 2 features

# Initialize and fit the PCA model
pca = PCA(n_components=2)  # You can specify the number of components you want to retain
pca.fit(data)

# Transform the data into the principal components
transformed_data = pca.transform(data)

# Explained variance ratio for each component
explained_variance_ratio = pca.explained_variance_ratio_
print("Explained Variance Ratio:", explained_variance_ratio)

# Scree plot to visualize explained variance
plt.bar(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio)
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance Ratio')
plt.title('Scree Plot')
plt.show()

# Principal Component 1 vs. Principal Component 2
plt.scatter(transformed_data[:, 0], transformed_data[:, 1])
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA: Principal Component 1 vs. Principal Component 2')
plt.show()
```

In this code:

1. We create a sample dataset with 100 samples and 2 features.
2. We initialize a PCA model, specifying the number of components to retain (in this case, 2).
3. We fit the PCA model to our data using the `fit` method.
4. We transform the original data into the principal components using the `transform` method.
5. We print the explained variance ratio, which tells you the proportion of variance explained by each principal component.
6. We create a scree plot to visualize the explained variance for each component.
7. We create a scatter plot to visualize the data in the principal component space.

9. Neural Networks Introduction:

1. Basic feedforward neural network for classification.

Python code example for a feedforward neural network used for classification. In this example, we'll use the popular deep learning framework Keras, which is often integrated with TensorFlow.

First, ensure you have Keras and TensorFlow installed. You can install them using pip:

```
pip install tensorflow
```

Now, here's a simple feedforward neural network code for classification:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical

# Sample data generation
# Replace this with your actual dataset
X = np.random.rand(100, 2)  # Sample features (100 samples, 2 features)
y = (X[:, 0] + X[:, 1] > 1).astype(int)  # A simple classification task

# One-hot encode the target labels (optional, depends on the problem)
y = to_categorical(y)

# Build the neural network model
model = Sequential()
model.add(Dense(8, input_dim=2, activation='relu'))
model.add(Dense(2, activation='softmax'))  # Two output classes, using softmax for classification

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
```

```
model.fit(X, y, epochs=100, batch_size=10)

# Evaluate the model
loss, accuracy = model.evaluate(X, y)
print(f'Loss: {loss}, Accuracy: {accuracy}')

# Make predictions
predictions = model.predict(X)
# You can now use these predictions for classification tasks
```

In this code:
1. We generate some sample data (replace this with your dataset).
2. We one-hot encode the target labels if your problem requires it (e.g., multi-class classification).
3. We create a Sequential neural network model with two dense (fully connected) layers.
4. We compile the model with the categorical cross-entropy loss and the Adam optimizer.
5. We train the model on the sample data.
6. We evaluate the model's performance on the same data.
7. Finally, we make predictions using the trained model.

2. Understanding activation functions and backpropagation.

Python code example that demonstrates the implementation of activation functions (specifically, the sigmoid and ReLU functions) and a simple backpropagation algorithm for a feedforward neural network with one hidden layer. This example is for educational purposes and doesn't include the entire training process with datasets, but it illustrates the key concepts.

```
import numpy as np

# Define the sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Define the derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)

# Define the ReLU (Rectified Linear Unit) activation function
def relu(x):
    return np.maximum(0, x)

# Define the derivative of the ReLU function
def relu_derivative(x):
    return np.where(x > 0, 1, 0)

# Define the neural network architecture
input_size = 2
hidden_size = 4
```

```
output_size = 1

# Initialize weights and biases
np.random.seed(0)
weights_input_hidden = np.random.uniform(size=(input_size, hidden_size))
weights_hidden_output = np.random.uniform(size=(hidden_size, output_size))
bias_hidden = np.zeros((1, hidden_size))
bias_output = np.zeros((1, output_size))

# Input data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

# Target output
y = np.array([[0], [1], [1], [0]])

# Training hyperparameters
learning_rate = 0.1
epochs = 10000

# Training the neural network
for epoch in range(epochs):
    # Forward propagation
    hidden_layer_input = np.dot(X, weights_input_hidden) + bias_hidden
    hidden_layer_output = relu(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
    output_layer_output = sigmoid(output_layer_input)

    # Compute the loss
    error = y - output_layer_output

    # Backpropagation
    d_output = error * sigmoid_derivative(output_layer_output)
    error_hidden_layer = d_output.dot(weights_hidden_output.T)
    d_hidden_layer = error_hidden_layer * relu_derivative(hidden_layer_output)

    # Update weights and biases
    weights_hidden_output += hidden_layer_output.T.dot(d_output) * learning_rate
    weights_input_hidden += X.T.dot(d_hidden_layer) * learning_rate
    bias_output += np.sum(d_output, axis=0, keepdims=True) * learning_rate
    bias_hidden += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate

# Testing the neural network
hidden_layer_input = np.dot(X, weights_input_hidden) + bias_hidden
hidden_layer_output = relu(hidden_layer_input)

output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
```

```
output_layer_output = sigmoid(output_layer_input)

print("Predicted Output:")
print(output_layer_output)
```

In this code:
- The `sigmoid` and `relu` functions define the sigmoid and ReLU activation functions, respectively, while `sigmoid_derivative` and `relu_derivative` provide their derivatives.
- We define the architecture of a neural network with one hidden layer.
- The network is trained using backpropagation with a learning rate and a specified number of epochs.
- Finally, the trained network is used to make predictions.

10. Deployment and Model Serialization:

1. Saving and loading trained models.

To save and load trained machine learning models in Python, you can use the `joblib` library (part of the Scikit-Learn ecosystem) or Python's built-in `pickle` module. Here's an example of how to save and load a trained model using both methods:

Using joblib:

```
# Import necessary libraries
import joblib
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris

# Load a sample dataset
data = load_iris()
X, y = data.data, data.target

# Create and train a model (Logistic Regression in this example)
model = LogisticRegression(max_iter=1000)
model.fit(X, y)

# Save the trained model to a file
joblib.dump(model, 'trained_model.joblib')

# Load the model from the file
loaded_model = joblib.load('trained_model.joblib')

# You can now use the loaded_model for predictions
```

Using pickle:

```
# Import necessary libraries
import pickle
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris

# Load a sample dataset
data = load_iris()
X, y = data.data, data.target

# Create and train a model (Logistic Regression in this example)
model = LogisticRegression(max_iter=1000)
model.fit(X, y)

# Save the trained model to a file
with open('trained_model.pkl', 'wb') as file:
    pickle.dump(model, file)

# Load the model from the file
with open('trained_model.pkl', 'rb') as file:
    loaded_model = pickle.load(file)

# You can now use the loaded_model for predictions
```

Both methods allow you to save your trained machine learning model to a file and then load it back for making predictions or further analysis.

2. Model deployment using Flask frameworks.

Here's a basic example of how to deploy a machine learning model using the Flask framework in Python. This code assumes you have a trained model and you want to create a simple web API for making predictions using that model. In this example, we'll create a web service that accepts input data via an HTTP POST request and returns model predictions.

Make sure you have Flask installed. If not, you can install it using `pip install flask`.

```
from flask import Flask, request, jsonify
import pickle  # We'll use this to load the trained model
import numpy as np

app = Flask(__name)

# Load the trained machine learning model
with open('your_trained_model.pkl', 'rb') as model_file:
    model = pickle.load(model_file)
```

```
@app.route('/predict', methods=['POST'])
def predict():
    try:
        # Get the input data from the request
        data = request.get_json()

        # Ensure that the input data is in the expected format
        if 'features' not in data:
            return jsonify({'error': 'Invalid input format'})

        # Extract the features from the input data
        features = data['features']

        # Convert features to a format that your model expects (e.g., NumPy array)
        features = np.array(features).reshape(1, -1)

        # Make predictions using the loaded model
        predictions = model.predict(features)

        # Return the predictions as JSON
        return jsonify({'predictions': predictions.tolist()})

    except Exception as e:
        return jsonify({'error': str(e)})

if __name__ == '__main__':
    app.run(debug=True)
```

In this code:

1. We import Flask and necessary libraries.
2. We load your pre-trained machine learning model using `pickle`. Replace `'your_trained_model.pkl'` with the actual path to your trained model file.
3. We define a Flask route `/predict` that listens for POST requests. It expects a JSON object with a key `'features'` containing the input data for prediction.
4. We extract the input features from the request, convert them to the format expected by your model (e.g., NumPy array), and make predictions.
5. We return the predictions as JSON.
6. If an exception occurs during the prediction process, we return an error message.
7. The web service runs using `app.run(debug=True)`.

You can test this service by sending a POST request to `http://localhost:5000/predict` with a JSON body containing your input data.

11. Deep Learning:

1. Introduction to deep learning with a simple neural network or CNN.

Here's an example of a simple feedforward neural network using Python and the popular deep learning library, TensorFlow, with the Keras API. Additionally, I'll provide an example of a Convolutional Neural Network (CNN) for image classification.

Simple Feedforward Neural Network:

```
import tensorflow as tf
from tensorflow import keras

# Define a simple feedforward neural network
model = keras.Sequential([
   keras.layers.Flatten(input_shape=(28, 28)),  # Input layer (flatten 28x28 images)
   keras.layers.Dense(128, activation='relu'),  # Hidden layer with 128 neurons and ReLU activation
   keras.layers.Dense(10, activation='softmax')  # Output layer with 10 neurons (for 10 classes) and softmax activation
])

# Compile the model
model.compile(optimizer='adam',  # Optimizer
          loss='sparse_categorical_crossentropy',  # Loss function for classification
          metrics=['accuracy'])  # Evaluation metric

# Load and preprocess your dataset (e.g., MNIST)
# Replace this with your data loading and preprocessing code

# Train the model
model.fit(train_images, train_labels, epochs=10)

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print("Test accuracy:", test_acc)
```

In the above code, we create a simple feedforward neural network using Keras. You should replace `train_images`, `train_labels`, `test_images`, and `test_labels` with your actual training and testing data.

Convolutional Neural Network (CNN):

Here's an example of a simple CNN for image classification using Keras:

```
import tensorflow as tf
from tensorflow import keras

# Define a simple CNN model
model = keras.Sequential([
   keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)), # Convolutional
layer
   keras.layers.MaxPooling2D((2, 2)),  # Max-pooling layer
   keras.layers.Flatten(),  # Flatten the feature maps
   keras.layers.Dense(128, activation='relu'),  # Fully connected hidden layer
   keras.layers.Dense(10, activation='softmax')  # Output layer
])

# Compile the model
model.compile(optimizer='adam',  # Optimizer
              loss='sparse_categorical_crossentropy',  # Loss function for classification
              metrics=['accuracy'])  # Evaluation metric

# Load and preprocess your image dataset (e.g., using the TensorFlow ImageDataGenerator)
# Replace this with your data loading and preprocessing code

# Train the model
model.fit(train_images, train_labels, epochs=10)

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print("Test accuracy:", test_acc)
```

In this CNN example, we use Conv2D layers for convolution, MaxPooling2D layers for pooling, and Dense layers for the fully connected part of the network. Replace `train_images`, `train_labels`, `test_images`, and `test_labels` with your actual image dataset.

12. Computer Vision:

1. Building a Basic image classification model and deploy it using pre-trained models or custom convolutional neural networks.

Building an image classification model for pneumonia detection using CT scan images involves several steps. Here, I'll provide a Python code example using a custom Convolutional Neural Network (CNN) and also mention how to deploy it. Please note that this is a simplified example, and for a real-world application, you'd need a more extensive dataset and possibly fine-tuning.

1. Data Preparation:

Before starting, ensure you have a dataset of CT scan images labeled as normal and pneumonia. The dataset should be split into training and testing sets.

2. Import Libraries:
You'll need libraries such as TensorFlow/Keras for model creation and scikit-learn for evaluation. Ensure you have them installed.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix
```

3. Load and Preprocess Data:

```
# Define data paths
train_data_dir = 'path_to_train_data'
test_data_dir = 'path_to_test_data'

# Data preprocessing and augmentation
train_datagen = ImageDataGenerator(rescale=1./255,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True)
test_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(train_data_dir,
                                                    target_size=(128, 128),
                                                    batch_size=32,
                                                    class_mode='binary')
test_generator = test_datagen.flow_from_directory(test_data_dir,
                                                  target_size=(128, 128),
                                                  batch_size=32,
                                                  class_mode='binary')
```

4. Build a CNN Model:

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(128, 128, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
```

```
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

5. Train the Model:

```
model.fit(train_generator, epochs=10, validation_data=test_generator)
```

6. Evaluate the Model:

```
# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(test_generator)
print(f"Test Accuracy: {test_accuracy}")

# Make predictions on test data
predictions = model.predict(test_generator)
predicted_classes = np.round(predictions)

# Generate classification report and confusion matrix
print(classification_report(test_generator.classes, predicted_classes))
print(confusion_matrix(test_generator.classes, predicted_classes))
```

7. Deployment:

To deploy the model, you can use frameworks like Flask or FastAPI to create a simple API for inference. Here's a simplified example using Flask:

```
from flask import Flask, request, jsonify
app = Flask(__name)
@app.route('/predict', methods=['POST'])
def predict_pneumonia():
    # Extract the image from the request and preprocess it
    image = preprocess_image(request.data)
    # Make a prediction using the trained model
    prediction = model.predict(image)
    # Return the prediction
    return jsonify({'pneumonia': float(prediction[0])})
if __name__ == '__main__':
    app.run()
```

In a real-world scenario, you'd need to set up a web server to host this Flask app. Users could send POST requests with images, and the app would return predictions. This is a simplified example, and in practice, you may need more complex preprocessing, a larger dataset, and possibly more advanced CNN architectures for better accuracy.