**Import Libraries**

python

import pandas as pd import numpy as np import matplotlib.pyplot as plt import seaborn as sns from sklearn.preprocessing import StandardScaler from sklearn.model\_selection import train\_test\_split, cross\_val\_score, GridSearchCV, learning\_curve from sklearn.ensemble import RandomForestRegressor from sklearn.linear\_model import LinearRegression from sklearn.tree import DecisionTreeRegressor from sklearn.metrics import mean\_squared\_error, r2\_score from sklearn.decomposition import PCA import joblib

* **pandas**: For data manipulation and analysis.
* **numpy**: For numerical operations, like arrays and mathematical functions.
* **matplotlib.pyplot**: For creating visualizations (charts, plots).
* **seaborn**: For statistical data visualization, built on top of matplotlib.
* **StandardScaler**: For scaling/normalizing numerical features.
* **train\_test\_split**: For splitting the data into training and testing sets.
* **cross\_val\_score**: For performing cross-validation on a model.
* **GridSearchCV**: For hyperparameter tuning of models.
* **learning\_curve**: To plot the learning curve for a model.
* **RandomForestRegressor, LinearRegression, DecisionTreeRegressor**: Machine learning models for regression tasks.
* **mean\_squared\_error, r2\_score**: Metrics for model evaluation.
* **PCA**: Principal Component Analysis for dimensionality reduction.
* **joblib**: For saving and loading models.

**Load Data from CSV Files**

python

df\_group1 = pd.read\_csv('FOOD-DATA-GROUP1.csv') df\_group2 = pd.read\_csv('FOOD-DATA-GROUP2.csv') df\_group3 = pd.read\_csv('FOOD-DATA-GROUP3.csv') df\_group4 = pd.read\_csv('FOOD-DATA-GROUP4.csv') df\_group5 = pd.read\_csv('FOOD-DATA-GROUP5.csv')

* **pd.read\_csv()**: Loads CSV files into pandas DataFrames.

python

df = pd.concat([df\_group1, df\_group2, df\_group3, df\_group4, df\_group5], axis=0, ignore\_index=True)

* **pd.concat()**: Concatenates the individual dataframes into a single dataframe (df). axis=0 indicates stacking them vertically (row-wise). ignore\_index=True resets the index.

**Explore the Data**

python

print(df.head())

* Displays the first few rows of the dataset.

python

print(df.info())

* Provides basic information about the dataset, such as the number of rows, columns, and the data types of each column.

python

print(df.describe())

* Provides statistical summaries of the numerical features (mean, standard deviation, min, max, etc.).

python

missing\_data = df.isnull().sum() print(f"Missing Data:\n{missing\_data[missing\_data > 0]}")

* **df.isnull().sum()**: Checks for missing values in the dataset and returns the count of missing values per column.
* The result is filtered to show only columns with missing data.

**Data Cleaning - Duplicates**

python

df.drop\_duplicates(inplace=True)

* Removes any duplicate rows from the DataFrame.

python

df.duplicated().sum()

* Checks if there are any remaining duplicates and returns the count.

**Feature Exploration**

python

df[['Fat', 'Protein', 'Carbohydrates']].hist(bins=20, figsize=(10, 8)) plt.show()

* **hist()**: Plots histograms for the selected columns (Fat, Protein, and Carbohydrates) to visualize their distributions.

**Detecting Outliers**

python

sns.boxplot(x=df['Fat']) plt.title('Outliers in Fat') plt.show()

* **sns.boxplot()**: Creates a boxplot for the Fat column to visually identify potential outliers.

**Correlation Matrix**

python

corr\_matrix = df.corr()

* **df.corr()**: Computes the correlation matrix between all numerical features in the dataset.

python

plt.figure(figsize=(10, 8)) sns.heatmap(corr\_matrix, annot=True, cmap='coolwarm', fmt='.2f') plt.title('Correlation Matrix of Nutritional Features') plt.show()

* **sns.heatmap()**: Creates a heatmap to visualize the correlation matrix. annot=True adds the correlation values to the heatmap, and cmap='coolwarm' sets the color scheme.

**Handle Non-Numeric Values in Numeric Columns**

python

numeric\_df = df.select\_dtypes(include=['float64', 'int64'])

* Selects only the numeric columns (float64 and int64 data types) from the DataFrame.

python

numeric\_df = numeric\_df.apply(pd.to\_numeric, errors='coerce')

* **pd.to\_numeric()**: Converts non-numeric values to NaN (errors='coerce').

python

numeric\_df = numeric\_df.fillna(numeric\_df.mean()) *# Fill NaN values with the mean of each column*

* Fills any missing (NaN) values with the mean of each respective column.

python

corr\_matrix = numeric\_df.corr()

* Recalculates the correlation matrix after handling the missing data.

**PCA for Dimensionality Reduction**

python

scaler = StandardScaler() scaled\_data = scaler.fit\_transform(df[['Caloric Value', 'Fat', 'Protein', 'Carbohydrates']])

* **StandardScaler()**: Scales the features so that they have zero mean and unit variance.
* **fit\_transform()**: Fits the scaler and transforms the data accordingly.

python

pca = PCA(n\_components=2) principal\_components = pca.fit\_transform(scaled\_data)

* **PCA(n\_components=2)**: Initializes PCA to reduce the data to 2 principal components.
* **fit\_transform()**: Fits PCA to the data and reduces its dimensionality.

python

pca\_df = pd.DataFrame(data=principal\_components, columns=['PC1', 'PC2'])

* Creates a new DataFrame with the 2 principal components.

python

plt.figure(figsize=(8, 6)) plt.scatter(pca\_df['PC1'], pca\_df['PC2'], alpha=0.7) plt.title('PCA: Dimensionality Reduction of Nutritional Features') plt.xlabel('Principal Component 1') plt.ylabel('Principal Component 2') plt.show()

* **scatter()**: Plots the 2 principal components (PC1 and PC2) on a 2D scatter plot.

**Train-Test Split**

python

X = df[['Fat', 'Protein', 'Carbohydrates']] y = df['Caloric Value']

* **X** contains the features (nutrients), and **y** contains the target variable (caloric value).

python

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

* **train\_test\_split()**: Splits the data into training and testing sets (80% train, 20% test).

**Model Training & Evaluation (Linear Regression, Decision Tree, Random Forest)**

For each of the models:

1. **Model Training**: The model is trained on the training set (X\_train, y\_train).
2. **Prediction**: The model makes predictions on the test set (X\_test).
3. **Evaluation**: The model’s performance is evaluated using the Root Mean Squared Error (RMSE) and R² score.

python

*# Linear Regression* lr\_model = LinearRegression() lr\_model.fit(X\_train, y\_train) y\_pred\_lr = lr\_model.predict(X\_test) lr\_rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred\_lr)) lr\_r2 = r2\_score(y\_test, y\_pred\_lr)

python

*# Decision Tree Regressor* dt\_model = DecisionTreeRegressor(random\_state=42) dt\_model.fit(X\_train, y\_train) y\_pred\_dt = dt\_model.predict(X\_test) dt\_rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred\_dt)) dt\_r2 = r2\_score(y\_test, y\_pred\_dt)

python

*# Random Forest Regressor* rf\_model = RandomForestRegressor(random\_state=42) rf\_model.fit(X\_train, y\_train) y\_pred\_rf = rf\_model.predict(X\_test) rf\_rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred\_rf)) rf\_r2 = r2\_score(y\_test, y\_pred\_rf)

**Model Comparison**

python

models\_rmse = {'Linear Regression': lr\_rmse, 'Decision Tree': dt\_rmse, 'Random Forest': rf\_rmse} sns.barplot(x=list(models\_rmse.keys()), y=list(models\_rmse.values()))

* Compares the RMSE of the different models using a barplot.

python

models\_r2 = {'Linear Regression': lr\_r2, 'Decision Tree': dt\_r2, 'Random Forest': rf\_r2} sns.barplot(x=list(models\_r2.keys()), y=list(models\_r2.values()))

* Compares the R² scores of the different models using a barplot.

**Hyperparameter Tuning with Grid Search**

python

*# Hyperparameter tuning for Random Forest using GridSearchCV* param\_grid = { 'n\_estimators': [50, 100, 200], 'max\_depth': [10, 20, None], 'min\_samples\_split': [2, 5, 10] } grid\_search = GridSearchCV(estimator=rf\_model, param\_grid=param\_grid, cv=5, scoring='neg\_mean\_squared\_error') grid\_search.fit(X\_train, y\_train) *# Get the best parameters and model* best\_rf\_model = grid\_search.best\_estimator\_ print(f'Best Hyperparameters: {grid\_search.best\_params\_}')

**Explanation:**

* **Hyperparameter Tuning:** This section is tuning the Random Forest model's hyperparameters (n\_estimators, max\_depth, min\_samples\_split) using GridSearchCV, which tests different combinations of hyperparameters to find the best set for the model.
* **param\_grid:** Specifies the grid of parameters to be tested.
* **GridSearchCV:** It is used to find the best parameters for the model by performing cross-validation (CV). The scoring='neg\_mean\_squared\_error' specifies that the mean squared error should be minimized.
* **grid\_search.fit():** This trains the model using all the possible hyperparameter combinations in the grid.
* **best\_rf\_model:** Once training is complete, best\_estimator\_ will give the model with the best combination of hyperparameters.
* **Print Best Parameters:** Outputs the best hyperparameters found during the grid search.

**Feature Importance in Random Forest**

python

*# Get feature importance from Random Forest model* feature\_importance = best\_rf\_model.feature\_importances\_ *# Visualize the feature importance* plt.figure(figsize=(8, 6)) sns.barplot(x=['Fat', 'Protein', 'Carbohydrates'], y=feature\_importance, palette='viridis') plt.title('Feature Importance in Random Forest') plt.show()

**Explanation:**

* **Feature Importance:** This extracts the importance of each feature from the trained Random Forest model. It helps in understanding which features contribute the most to the model’s predictions.
* **Visualization:** A bar plot is used to visualize the importance of features (Fat, Protein, and Carbohydrates) based on their importance values.

**Save the Trained Model**

python

*# Save the trained Random Forest model* joblib.dump(best\_rf\_model, 'food\_nutritional\_model\_rf.pkl')

**Explanation:**

* **Save Model:** This line uses joblib.dump() to save the trained Random Forest model (best\_rf\_model) to a file (food\_nutritional\_model\_rf.pkl) for future use. The model can be loaded later for prediction without needing to retrain.

**Save Scaler**

python

*# Save the scaler* joblib.dump(scaler, 'scaler.pkl')

**Explanation:**

* **Save Scaler:** Similar to the model, the StandardScaler is saved to a file (scaler.pkl). This is done so the scaling transformation can be applied to new data when the model is used in the future (i.e., scaling the features using the same parameters as in training).

**Load the Saved Model**

python

*# Load the trained Random Forest model* loaded\_rf\_model = joblib.load('food\_nutritional\_model\_rf.pkl') *# Make predictions using the loaded model* y\_pred\_loaded = loaded\_rf\_model.predict(X\_test)

**Explanation:**

* **Load Model:** This loads the previously saved Random Forest model using joblib.load(). This is useful for making predictions on new data without needing to retrain the model.
* **Make Predictions:** After loading the model, predictions (y\_pred\_loaded) are made on the test set (X\_test).

**Learning Curve for Random Forest**

python

*# Plot learning curve for Random Forest model* train\_sizes, train\_scores, test\_scores = learning\_curve( estimator=best\_rf\_model, X=X\_train, y=y\_train, cv=5, scoring='neg\_mean\_squared\_error', train\_sizes=np.linspace(0.1, 1.0, 10) ) *# Calculate average training and test scores* train\_mean = np.mean(-train\_scores, axis=1) test\_mean = np.mean(-test\_scores, axis=1) plt.figure(figsize=(8, 6)) plt.plot(train\_sizes, train\_mean, label='Training Error', color='blue') plt.plot(train\_sizes, test\_mean, label='Cross-validation Error', color='red') plt.title('Learning Curve for Random Forest') plt.xlabel('Training Size') plt.ylabel('RMSE') plt.legend() plt.show()

**Explanation:**

* **Learning Curve:** This section generates a learning curve to visualize how the model’s error decreases as the amount of training data increases.
* learning\_curve()**Function:** This function from sklearn.model\_selection computes training and cross-validation scores over a range of training data sizes (specified by train\_sizes).
* **Plotting:** It plots both training error and cross-validation error on the same graph to see if the model is overfitting (i.e., the training error is much lower than the cross-validation error).

**Residual Plot for Random Forest**

python

*# Plot residuals for Random Forest model* residuals = y\_test - y\_pred\_rf plt.figure(figsize=(8, 6)) sns.scatterplot(x=y\_pred\_rf, y=residuals, color='purple') plt.axhline(y=0, color='r', linestyle='--') plt.title('Residuals for Random Forest Model') plt.xlabel('Predicted Values') plt.ylabel('Residuals') plt.show()

**Explanation:**

* **Residuals Plot:** Residuals are the differences between the true values (y\_test) and the predicted values (y\_pred\_rf). A residual plot helps visualize if the model has any systematic errors.
* **Plotting:** A scatter plot is used to visualize residuals versus predicted values. If the model is good, the residuals should be randomly scattered around zero (indicating no pattern).

**Evaluating Multicollinearity with VIF (Variance Inflation Factor)**

python

from statsmodels.stats.outliers\_influence import variance\_inflation\_factor from statsmodels.tools.tools import add\_constant *# Add constant to X (for VIF calculation)* X\_with\_const = add\_constant(X) *# Calculate VIF for each feature* vif\_data = pd.DataFrame() vif\_data['Feature'] = X\_with\_const.columns vif\_data['VIF'] = [variance\_inflation\_factor(X\_with\_const.values, i) for i in range(X\_with\_const.shape[1])] *# Display VIF values* print(vif\_data)

**Explanation:**

* **VIF (Variance Inflation Factor):** This is used to check for multicollinearity among the features. Multicollinearity occurs when two or more features are highly correlated, which can distort the model's coefficients.
* **VIF Calculation:** The code adds a constant column to the feature matrix (X\_with\_const) and calculates the VIF for each feature using variance\_inflation\_factor(). The result is stored in a DataFrame and printed.
* **VIF Interpretation:** A VIF above 10 indicates high multicollinearity and suggests that a feature might need to be removed.

**Feature Scaling (Standardization)**

python

*# Standardize the features for better model performance* scaler = StandardScaler() df\_scaled = scaler.fit\_transform(df[['Fat', 'Protein', 'Carbohydrates', 'Caloric Value']]) *# Verify scaling by checking mean and standard deviation* print('Mean after scaling:', df\_scaled.mean(axis=0)) print('Standard deviation after scaling:', df\_scaled.std(axis=0))

**Explanation:**

* **Feature Scaling:** This section standardizes the features by scaling them to have zero mean and unit variance using StandardScaler(). Standardization improves the performance of some models (like SVM or Logistic Regression).
* **Checking Scaling:** After scaling, it prints the mean and standard deviation of the scaled features. The mean should be close to zero, and the standard deviation should be close to one.

**Train a Support Vector Machine Model**

python

from sklearn.svm import SVR *# Train a Support Vector Machine model* svm\_model = SVR(kernel='rbf') svm\_model.fit(X\_train, y\_train) *# Predict using the SVM model* y\_pred\_svm = svm\_model.predict(X\_test) *# Evaluate performance of SVM model* svm\_rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred\_svm)) svm\_r2 = r2\_score(y\_test, y\_pred\_svm) print(f'Support Vector Machine RMSE: {svm\_rmse:.2f}') print(f'Support Vector Machine R2: {svm\_r2:.2f}')

**Explanation:**

* **SVR (Support Vector Regression):** This section trains a Support Vector Machine (SVM) regression model (SVR) to predict the target variable (Caloric Value). The kernel='rbf' specifies the use of the Radial Basis Function (RBF) kernel.
* **Model Training:** svm\_model.fit() trains the model using the training data.
* **Model Evaluation:** Predictions are made using svm\_model.predict(), and performance is evaluated using RMSE and R² metrics.

**Visualizing Model Performance with Prediction vs Actual Plot**

python

Copy code

*# Plot Actual vs Predicted values for Random Forest* plt.figure(figsize=(8, 6)) plt.scatter(y\_test, y\_pred\_rf, color='green', alpha=0.6) plt.plot([min(y\_test), max(y\_test)], [min(y\_test), max(y\_test)], color='red', linestyle='--') plt.title('Actual vs Predicted Values for Random Forest') plt.xlabel('Actual Values') plt.ylabel('Predicted Values') plt.show()

**Explanation:**

* **Actual vs Predicted Plot:** This section visualizes the performance of the Random Forest model by plotting the actual vs predicted values. It helps in assessing how close the predictions are to the true values.
* **Scatter Plot:** The actual values (y\_test) are plotted on the x-axis, and the predicted values (y\_pred\_rf) are on the y-axis. A red dashed line is also drawn that represents perfect predictions, where the predicted value equals the actual value.
* **Plot Interpretation:** The closer the green points (predictions) are to the red dashed line, the better the model's predictions.

**Model Comparison: Random Forest vs Support Vector Machine**

python

Copy code

*# Plot RMSE for Random Forest and Support Vector Machine models* rmse\_rf = np.sqrt(mean\_squared\_error(y\_test, y\_pred\_rf)) rmse\_svm = np.sqrt(mean\_squared\_error(y\_test, y\_pred\_svm)) plt.figure(figsize=(8, 6)) plt.bar(['Random Forest', 'SVM'], [rmse\_rf, rmse\_svm], color=['blue', 'orange']) plt.title('Model Comparison (RMSE)') plt.ylabel('Root Mean Squared Error (RMSE)') plt.show()

**Explanation:**

* **Model Comparison:** This compares the performance of the Random Forest and Support Vector Machine models by plotting their RMSE (Root Mean Squared Error) values.
* **Bar Plot:** The RMSE values for both models are displayed as bars. The lower the RMSE, the better the model's performance.
* **Plot Interpretation:** This visualization helps in comparing which model performs better in terms of prediction accuracy.

**Model Performance Metrics (R² and RMSE)**

python

Copy code

*# Print the performance metrics for both models* print(f'Random Forest RMSE: {rmse\_rf:.2f}') print(f'Support Vector Machine RMSE: {rmse\_svm:.2f}') print(f'Random Forest R²: {r2\_score(y\_test, y\_pred\_rf):.2f}') print(f'Support Vector Machine R²: {r2\_score(y\_test, y\_pred\_svm):.2f}')

**Explanation:**

* **Performance Metrics:** This section prints the performance metrics for both models, namely RMSE (Root Mean Squared Error) and R² (coefficient of determination).
* **RMSE:** This metric represents the average magnitude of the errors between predicted and actual values. A lower RMSE indicates better performance.
* **R²:** This metric tells you how well the model's predictions match the actual values. An R² of 1 indicates perfect predictions, and values closer to 0 indicate poor performance.