**import pandas as pd**

**import numpy as np**

**import matplotlib.pyplot as plt**

**import seaborn as sns**

**from pymatgen.core.composition import Composition**

**from matminer.featurizers.composition import ElementProperty, Meredig, OxidationStates**

**from sklearn.model\_selection import train\_test\_split, cross\_val\_score, KFold**

**from sklearn.ensemble import GradientBoostingRegressor, RandomForestRegressor**

**from sklearn.metrics import mean\_absolute\_error, r2\_score, make\_scorer**

**from sklearn.preprocessing import StandardScaler**

**from sklearn.impute import SimpleImputer**

**from sklearn.svm import SVR**

**from sklearn.linear\_model import LinearRegression, Ridge, Lasso, ElasticNet**

**from sklearn.neighbors import KNeighborsRegressor**

**from xgboost import XGBRegressor**

**from lightgbm import LGBMRegressor**

**import joblib**

**import warnings**

**from tqdm import tqdm**

**import torch**

**import torch.nn as nn**

**import torch.optim as optim**

**from torch.utils.data import DataLoader, TensorDataset**

**from sklearn.ensemble import StackingRegressor, VotingRegressor**

**# Suppress warnings for cleaner output**

**warnings.filterwarnings('ignore')**

**# ======================**

**# DEEP LEARNING MODELS (PyTorch Implementation)**

**# ======================**

**class MLPRegressor(nn.Module):**

**"""Base MLP Regressor"""**

**def \_\_init\_\_(self, input\_dim, layer\_sizes, activation='relu', dropout\_rate=0.2):**

**super(MLPRegressor, self).\_\_init\_\_()**

**layers = []**

**prev\_size = input\_dim**

**for size in layer\_sizes[:-1]:**

**layers.append(nn.Linear(prev\_size, size))**

**if activation == 'relu':**

**layers.append(nn.ReLU())**

**elif activation == 'leakyrelu':**

**layers.append(nn.LeakyReLU(0.1))**

**elif activation == 'tanh':**

**layers.append(nn.Tanh())**

**layers.append(nn.Dropout(dropout\_rate))**

**prev\_size = size**

**layers.append(nn.Linear(prev\_size, layer\_sizes[-1]))**

**self.layers = nn.Sequential(\*layers)**

**def forward(self, x):**

**return self.layers(x)**

**def fit(self, X, y, epochs=1000, batch\_size=32, lr=0.001, verbose=False):**

**X\_tensor = torch.tensor(X, dtype=torch.float32)**

**y\_tensor = torch.tensor(y, dtype=torch.float32).view(-1, 1)**

**dataset = TensorDataset(X\_tensor, y\_tensor)**

**loader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True)**

**criterion = nn.L1Loss()**

**optimizer = optim.Adam(self.parameters(), lr=lr)**

**# Early stopping setup**

**best\_loss = float('inf')**

**patience = 50**

**patience\_counter = 0**

**for epoch in range(epochs):**

**self.train()**

**epoch\_loss = 0.0**

**for inputs, targets in loader:**

**optimizer.zero\_grad()**

**outputs = self(inputs)**

**loss = criterion(outputs, targets)**

**loss.backward()**

**optimizer.step()**

**epoch\_loss += loss.item()**

**# Validation**

**self.eval()**

**with torch.no\_grad():**

**val\_pred = self(X\_tensor)**

**val\_loss = criterion(val\_pred, y\_tensor).item()**

**if verbose and epoch % 100 == 0:**

**print(f'Epoch {epoch}: Train Loss {epoch\_loss/len(loader):.4f}, Val Loss {val\_loss:.4f}')**

**# Early stopping**

**if val\_loss < best\_loss:**

**best\_loss = val\_loss**

**patience\_counter = 0**

**best\_state = self.state\_dict()**

**else:**

**patience\_counter += 1**

**if patience\_counter >= patience:**

**if verbose:**

**print(f'Early stopping at epoch {epoch}')**

**self.load\_state\_dict(best\_state)**

**break**

**return self**

**def predict(self, X):**

**self.eval()**

**with torch.no\_grad():**

**X\_tensor = torch.tensor(X, dtype=torch.float32)**

**preds = self(X\_tensor).numpy().flatten()**

**return preds**

**class MLP1(MLPRegressor):**

**"""Standard MLP (128-64-32)"""**

**def \_\_init\_\_(self, input\_dim):**

**super().\_\_init\_\_(input\_dim, [128, 64, 32, 1], 'relu', 0.2)**

**class MLP2(MLPRegressor):**

**"""Deeper MLP (256-128-64-32)"""**

**def \_\_init\_\_(self, input\_dim):**

**super().\_\_init\_\_(input\_dim, [256, 128, 64, 32, 1], 'relu', 0.3)**

**class MLP3(MLPRegressor):**

**"""Wider MLP (256-256)"""**

**def \_\_init\_\_(self, input\_dim):**

**super().\_\_init\_\_(input\_dim, [256, 256, 1], 'leakyrelu', 0.2)**

**class MLP4(MLPRegressor):**

**"""Batch Normalization MLP (128-64)"""**

**def \_\_init\_\_(self, input\_dim):**

**super().\_\_init\_\_(input\_dim, [128, 64, 1], 'relu', 0.2)**

**# Add batch normalization**

**self.bn1 = nn.BatchNorm1d(128)**

**self.bn2 = nn.BatchNorm1d(64)**

**def forward(self, x):**

**x = self.layers[0](x) # First linear layer**

**x = self.bn1(x)**

**x = self.layers[1](x) # ReLU**

**x = self.layers[2](x) # Dropout**

**x = self.layers[3](x) # Second linear**

**x = self.bn2(x)**

**x = self.layers[4](x) # ReLU**

**x = self.layers[5](x) # Dropout**

**x = self.layers[6](x) # Final linear**

**return x**

**class MLP5(MLPRegressor):**

**"""Tanh Activation MLP (128-64)"""**

**def \_\_init\_\_(self, input\_dim):**

**super().\_\_init\_\_(input\_dim, [128, 64, 1], 'tanh', 0.1)**

**# ======================**

**# DATA PREPARATION**

**# ======================**

**# Load periodic table data**

**periodic\_table = pd.read\_csv(r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\Periodic Table of Elements.csv", encoding='latin1')**

**# Clean and prepare periodic table data**

**periodic\_table = periodic\_table.rename(columns={'Symbol': 'symbol'})**

**periodic\_table = periodic\_table.drop\_duplicates('symbol') # Ensure one entry per element**

**# List of material dataset filenames**

**files = [**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\Ba based material cubic.csv",**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\Be based material cubic.csv",**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\Ca cubic based material.csv",**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\Cd based material cubic.csv",**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\Hg based material cubic.csv",**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\Mg based material cubic.csv",**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\O based material cubic.csv",**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\S based material cubic.csv",**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\Se based material cubic.csv",**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\Sr based material cubic.csv",**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\Te based material cubic.csv",**

**r"C:\Users\Acer\Desktop\Ca group and chalcogenide cubic group data2\Zn based materials.csv"**

**]**

**# Load and combine datasets**

**dfs = []**

**for f in files:**

**df = pd.read\_csv(f)**

**# Standardize band gap column name**

**if 'Band gap' in df.columns:**

**df = df.rename(columns={'Band gap': 'band\_gap'})**

**if 'Band Gap' in df.columns:**

**df = df.rename(columns={'Band Gap': 'band\_gap'})**

**dfs.append(df)**

**materials = pd.concat(dfs, ignore\_index=True)**

**print(f"Combined dataset: {materials.shape[0]} entries")**

**# Create composition objects**

**materials['composition'] = materials['Formula'].apply(Composition)**

**# ======================**

**# FEATURE ENGINEERING**

**# ======================**

**# Initialize featurizers**

**featurizer = ElementProperty.from\_preset('magpie')**

**meredig\_featurizer = Meredig()**

**oxidation\_featurizer = OxidationStates()**

**# Generate composition-based features**

**print("Generating composition features...")**

**composition\_features = featurizer.featurize\_dataframe(materials, 'composition', ignore\_errors=True)**

**composition\_features = meredig\_featurizer.featurize\_dataframe(composition\_features, 'composition', ignore\_errors=True)**

**composition\_features = oxidation\_featurizer.featurize\_dataframe(composition\_features, 'composition', ignore\_errors=True)**

**# Extract feature columns**

**feature\_labels = (**

**featurizer.feature\_labels() +**

**meredig\_featurizer.feature\_labels() +**

**oxidation\_featurizer.feature\_labels()**

**)**

**def add\_periodic\_features(df, periodic\_table):**

**"""Add weighted average elemental properties from the periodic table to each composition."""**

**print("Adding periodic table features...")**

**# Define the properties we want to include**

**periodic\_props = [**

**'AtomicMass', 'AtomicRadius', 'Electronegativity',**

**'FirstIonization', 'Density', 'MeltingPoint',**

**'BoilingPoint', 'NumberOfIsotopes', 'SpecificHeat',**

**'NumberofValence', 'NumberofShells'**

**]**

**# Create dictionary of element properties**

**element\_props = periodic\_table.set\_index('symbol')[periodic\_props].to\_dict('index')**

**# Initialize lists for weighted features**

**weighted\_features = {prop: [] for prop in periodic\_props}**

**# Process each composition**

**for comp in tqdm(df['composition']):**

**# Get element amounts dictionary**

**try:**

**if isinstance(comp, str):**

**comp = Composition(comp)**

**el\_amt\_dict = comp.get\_el\_amt\_dict()**

**except Exception as e:**

**print(f"Error processing composition: {comp}")**

**el\_amt\_dict = {}**

**total\_weight = sum(el\_amt\_dict.values()) if el\_amt\_dict else 0**

**prop\_sums = {prop: 0.0 for prop in periodic\_props}**

**for el\_symbol, amt in el\_amt\_dict.items():**

**if el\_symbol not in element\_props:**

**continue**

**for prop in periodic\_props:**

**# Handle missing values by using 0**

**val = element\_props[el\_symbol].get(prop, 0)**

**if pd.isna(val):**

**val = 0**

**try:**

**prop\_sums[prop] += float(val) \* float(amt)**

**except:**

**continue**

**for prop in periodic\_props:**

**weighted\_avg = prop\_sums[prop] / total\_weight if total\_weight > 0 else np.nan**

**weighted\_features[prop].append(weighted\_avg)**

**# Add the new features to the dataframe**

**for prop in periodic\_props:**

**df[f'periodic\_{prop}'] = weighted\_features[prop]**

**return df, periodic\_props**

**# Add periodic table features**

**composition\_features, periodic\_props = add\_periodic\_features(composition\_features, periodic\_table)**

**# Update feature labels with periodic table features**

**feature\_labels += [f'periodic\_{prop}' for prop in periodic\_props]**

**# Prepare feature matrix and target**

**X = composition\_features[feature\_labels]**

**y = materials['band\_gap'].values**

**# Handle missing values**

**imputer = SimpleImputer(strategy='median')**

**X\_imputed = imputer.fit\_transform(X)**

**# Train-test split**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(**

**X\_imputed, y, test\_size=0.2, random\_state=42**

**)**

**# Scaling**

**scaler = StandardScaler()**

**X\_train\_scaled = scaler.fit\_transform(X\_train)**

**X\_test\_scaled = scaler.transform(X\_test)**

**# ======================**

**# MODEL TRAINING WITH 10-FOLD CROSS-VALIDATION**

**# ======================**

**# Define all models to train including multiple MLPs**

**models = {**

**'Gradient Boosting': GradientBoostingRegressor(**

**learning\_rate=0.05,**

**max\_depth=5,**

**n\_estimators=300,**

**subsample=0.8,**

**random\_state=42**

**),**

**'Random Forest': RandomForestRegressor(**

**n\_estimators=200,**

**max\_depth=10,**

**random\_state=42**

**),**

**'Support Vector': SVR(kernel='rbf', C=10, gamma='scale'),**

**'Linear Regression': LinearRegression(),**

**'Ridge Regression': Ridge(alpha=1.0),**

**'Lasso Regression': Lasso(alpha=0.1),**

**'ElasticNet': ElasticNet(alpha=0.1, l1\_ratio=0.5),**

**'K-Neighbors': KNeighborsRegressor(n\_neighbors=5),**

**'XGBoost': XGBRegressor(**

**learning\_rate=0.05,**

**max\_depth=5,**

**n\_estimators=300,**

**subsample=0.8,**

**random\_state=42**

**),**

**'LightGBM': LGBMRegressor(**

**learning\_rate=0.05,**

**max\_depth=5,**

**n\_estimators=300,**

**subsample=0.8,**

**random\_state=42**

**),**

**'MLP1': MLP1(input\_dim=X\_train\_scaled.shape[1]),**

**'MLP2': MLP2(input\_dim=X\_train\_scaled.shape[1]),**

**'MLP3': MLP3(input\_dim=X\_train\_scaled.shape[1]),**

**'MLP4': MLP4(input\_dim=X\_train\_scaled.shape[1]),**

**'MLP5': MLP5(input\_dim=X\_train\_scaled.shape[1])**

**}**

**# Initialize KFold**

**kfold = KFold(n\_splits=10, shuffle=True, random\_state=42)**

**mae\_scorer = make\_scorer(mean\_absolute\_error, greater\_is\_better=False)**

**# Train and evaluate all models with cross-validation**

**results = []**

**cv\_results = []**

**print("\nTraining and evaluating models with 10-fold CV...")**

**for name, model in tqdm(models.items(), desc="Models"):**

**print(f"\nEvaluating {name} with 10-fold CV...")**

**# Handle PyTorch models separately**

**if 'MLP' in name:**

**# Manual cross-validation for PyTorch models**

**mae\_scores = []**

**r2\_scores = []**

**for fold, (train\_idx, val\_idx) in enumerate(kfold.split(X\_train\_scaled)):**

**print(f" Fold {fold+1}/10")**

**X\_train\_fold, X\_val\_fold = X\_train\_scaled[train\_idx], X\_train\_scaled[val\_idx]**

**y\_train\_fold, y\_val\_fold = y\_train[train\_idx], y\_train[val\_idx]**

**# Reinitialize model for each fold**

**model\_instance = model.\_\_class\_\_(input\_dim=X\_train\_scaled.shape[1])**

**model\_instance.fit(X\_train\_fold, y\_train\_fold, verbose=False)**

**y\_val\_pred = model\_instance.predict(X\_val\_fold)**

**mae = mean\_absolute\_error(y\_val\_fold, y\_val\_pred)**

**r2 = r2\_score(y\_val\_fold, y\_val\_pred)**

**mae\_scores.append(mae)**

**r2\_scores.append(r2)**

**cv\_mae = np.mean(mae\_scores)**

**cv\_mae\_std = np.std(mae\_scores)**

**cv\_r2 = np.mean(r2\_scores)**

**cv\_r2\_std = np.std(r2\_scores)**

**else:**

**# Cross-validation for sklearn models**

**cv\_mae = -np.mean(cross\_val\_score(model, X\_train\_scaled, y\_train,**

**cv=kfold, scoring=mae\_scorer))**

**cv\_mae\_std = np.std(-cross\_val\_score(model, X\_train\_scaled, y\_train,**

**cv=kfold, scoring=mae\_scorer))**

**cv\_r2 = np.mean(cross\_val\_score(model, X\_train\_scaled, y\_train,**

**cv=kfold, scoring='r2'))**

**cv\_r2\_std = np.std(cross\_val\_score(model, X\_train\_scaled, y\_train,**

**cv=kfold, scoring='r2'))**

**# Save CV results**

**cv\_results.append({**

**'Model': name,**

**'CV\_MAE': cv\_mae,**

**'CV\_MAE\_std': cv\_mae\_std,**

**'CV\_R2': cv\_r2,**

**'CV\_R2\_std': cv\_r2\_std**

**})**

**print(f"{name} CV Performance:")**

**print(f"MAE: {cv\_mae:.4f} ± {cv\_mae\_std:.4f}, R²: {cv\_r2:.4f} ± {cv\_r2\_std:.4f}")**

**# Train on full training set**

**print(f"Training {name} on full training set...")**

**if 'MLP' in name:**

**model.fit(X\_train\_scaled, y\_train, verbose=False)**

**else:**

**model.fit(X\_train\_scaled, y\_train)**

**# Evaluate on test set**

**y\_pred = model.predict(X\_test\_scaled)**

**test\_mae = mean\_absolute\_error(y\_test, y\_pred)**

**test\_r2 = r2\_score(y\_test, y\_pred)**

**print(f"{name} Test Performance:")**

**print(f"MAE: {test\_mae:.4f}, R²: {test\_r2:.4f}")**

**# Save results**

**results.append({**

**'Model': name,**

**'Test\_MAE': test\_mae,**

**'Test\_R2': test\_r2**

**})**

**# Save model**

**if 'MLP' in name:**

**torch.save(model.state\_dict(), f"{name}\_model.pth")**

**else:**

**joblib.dump(model, f"{name.replace(' ', '\_')}\_model.joblib")**

**print(f"Saved {name} model")**

**# Create results dataframes**

**cv\_results\_df = pd.DataFrame(cv\_results)**

**results\_df = pd.DataFrame(results)**

**final\_results\_df = pd.merge(cv\_results\_df, results\_df, on='Model')**

**print("\nModel Performance Summary with Cross-Validation:")**

**print(final\_results\_df.sort\_values(by='CV\_MAE'))**

**# Save performance results**

**final\_results\_df.to\_csv("model\_performance\_with\_cv.csv", index=False)**

**# ======================**

**# CREATE TOP-4 ENSEMBLE (Based on CV performance)**

**# ======================**

**# Select top 4 models based on CV MAE**

**top\_4\_models = final\_results\_df.sort\_values(by='CV\_MAE').head(4)['Model'].tolist()**

**print(f"\nTop 4 models for ensemble: {top\_4\_models}")**

**# Create ensemble of top 4 models**

**class Top4Ensemble:**

**def \_\_init\_\_(self, model\_names, models\_dict):**

**self.model\_names = model\_names**

**self.models = []**

**# Load models**

**for name in model\_names:**

**if 'MLP' in name:**

**# For PyTorch models, we need to reconstruct and load state**

**model\_class = globals()[name]**

**model = model\_class(input\_dim=X\_train\_scaled.shape[1])**

**model.load\_state\_dict(torch.load(f"{name}\_model.pth"))**

**else:**

**model = joblib.load(f"{name.replace(' ', '\_')}\_model.joblib")**

**self.models.append((name, model))**

**def predict(self, X):**

**predictions = []**

**for name, model in self.models:**

**if 'MLP' in name:**

**# PyTorch models need to be in eval mode**

**model.eval()**

**with torch.no\_grad():**

**X\_tensor = torch.tensor(X, dtype=torch.float32)**

**pred = model(X\_tensor).numpy().flatten()**

**else:**

**pred = model.predict(X)**

**predictions.append(pred)**

**# Average predictions**

**return np.mean(predictions, axis=0)**

**# Initialize and save ensemble**

**ensemble = Top4Ensemble(top\_4\_models, models)**

**joblib.dump(ensemble, 'top4\_ensemble.joblib')**

**print("Saved top 4 ensemble model")**

**# Add ensemble to models dictionary for prediction**

**models['Top4\_Ensemble'] = ensemble**

**# Evaluate ensemble on test set**

**y\_pred\_ensemble = ensemble.predict(X\_test\_scaled)**

**ensemble\_mae = mean\_absolute\_error(y\_test, y\_pred\_ensemble)**

**ensemble\_r2 = r2\_score(y\_test, y\_pred\_ensemble)**

**print(f"Ensemble Test Performance: MAE = {ensemble\_mae:.4f}, R² = {ensemble\_r2:.4f}")**

**# ======================**

**# PREDICT TARGET COMPOSITIONS (ALL MODELS)**

**# ======================**

**target\_compositions = [**

**"HgTe", "Hg0.75Ca0.25Te", "Hg0.5Ca0.5Te", "Hg0.25Ca0.75Te", "CaTe",**

**]**

**def predict\_bandgaps(compositions, feature\_labels, imputer, scaler, models):**

**"""Predict band gaps using all trained models"""**

**pred\_df = pd.DataFrame({'composition': compositions})**

**pred\_df['comp\_obj'] = pred\_df['composition'].apply(Composition)**

**# Generate features using the same featurizers as training**

**X\_pred = featurizer.featurize\_dataframe(pred\_df, 'comp\_obj', ignore\_errors=True)**

**X\_pred = meredig\_featurizer.featurize\_dataframe(X\_pred, 'comp\_obj', ignore\_errors=True)**

**X\_pred = oxidation\_featurizer.featurize\_dataframe(X\_pred, 'comp\_obj', ignore\_errors=True)**

**# Add periodic table features**

**X\_pred, \_ = add\_periodic\_features(X\_pred, periodic\_table)**

**# Ensure feature alignment**

**X\_pred = X\_pred.reindex(columns=feature\_labels, fill\_value=np.nan)**

**# Preprocessing**

**X\_pred\_imputed = imputer.transform(X\_pred)**

**X\_pred\_scaled = scaler.transform(X\_pred\_imputed)**

**# Make predictions with all models**

**predictions = pd.DataFrame({'Composition': compositions})**

**for name, model in models.items():**

**if name == 'Top4\_Ensemble':**

**# Ensemble model**

**preds = model.predict(X\_pred\_scaled)**

**elif 'MLP' in name:**

**# Load PyTorch model for prediction**

**model.eval()**

**with torch.no\_grad():**

**X\_tensor = torch.tensor(X\_pred\_scaled, dtype=torch.float32)**

**preds = model(X\_tensor).numpy().flatten()**

**else:**

**preds = model.predict(X\_pred\_scaled)**

**predictions[name] = np.round(preds, 3)**

**return predictions**

**# Predict band gaps with all models**

**print("\nPredicting band gaps for target compositions...")**

**all\_predictions = predict\_bandgaps(**

**target\_compositions,**

**feature\_labels,**

**imputer,**

**scaler,**

**models**

**)**

**# Save predictions**

**all\_predictions.to\_csv("all\_models\_predictionsfor5composition.csv", index=False)**

**# Format for LaTeX**

**latex\_df = all\_predictions.copy()**

**latex\_df.columns = ['Composition'] + [col.replace(' ', '\n') for col in latex\_df.columns[1:]]**

**latex\_df.to\_latex(**

**"all\_models\_table.tex",**

**index=False,**

**caption="Predicted Band Gaps (All Models)",**

**label="tab:all\_bandgaps",**

**column\_format='l' + 'c' \* (len(latex\_df.columns) - 1),**

**escape=False**

**)**

**print("\nPredicted Band Gaps (All Models):")**

**print(all\_predictions)**

**print("\nResults saved to:")**

**print("- all\_models\_predictions.csv")**

**print("- all\_models\_table.tex")**

**print("- model\_performance.csv")**

**print("- Feature importance plots for tree-based models")**

**# ======================**

**# ENSEMBLE MODELS (Traditional)**

**# ======================**

**# Load the best individual models (or use the ones already in memory)**

**lgbm = joblib.load('LightGBM\_model.joblib')**

**rf = joblib.load('Random\_Forest\_model.joblib')**

**xgb = joblib.load('XGBoost\_model.joblib')**

**# 1. Simple Averaging Ensemble**

**print("\nEvaluating Simple Averaging Ensemble...")**

**lgbm\_pred = lgbm.predict(X\_test\_scaled)**

**rf\_pred = rf.predict(X\_test\_scaled)**

**xgb\_pred = xgb.predict(X\_test\_scaled)**

**avg\_pred = (lgbm\_pred + rf\_pred + xgb\_pred) / 3.0**

**avg\_mae = mean\_absolute\_error(y\_test, avg\_pred)**

**avg\_r2 = r2\_score(y\_test, avg\_pred)**

**print(f"Simple Averaging: MAE = {avg\_mae:.4f}, R² = {avg\_r2:.4f}")**

**# 2. Voting Regressor**

**print("\nTraining Voting Regressor...")**

**voting = VotingRegressor(**

**estimators=[**

**('lgbm', LGBMRegressor(learning\_rate=0.05, max\_depth=5,**

**n\_estimators=300, subsample=0.8, random\_state=42)),**

**('rf', RandomForestRegressor(n\_estimators=200, max\_depth=10, random\_state=42)),**

**('xgb', XGBRegressor(learning\_rate=0.05, max\_depth=5,**

**n\_estimators=300, subsample=0.8, random\_state=42))**

**]**

**)**

**voting.fit(X\_train\_scaled, y\_train)**

**voting\_pred = voting.predict(X\_test\_scaled)**

**voting\_mae = mean\_absolute\_error(y\_test, voting\_pred)**

**voting\_r2 = r2\_score(y\_test, voting\_pred)**

**print(f"Voting Regressor: MAE = {voting\_mae:.4f}, R² = {voting\_r2:.4f}")**

**# 3. Stacking Regressor**

**print("\nTraining Stacking Regressor...")**

**stacking = StackingRegressor(**

**estimators=[**

**('lgbm', LGBMRegressor(learning\_rate=0.05, max\_depth=5,**

**n\_estimators=300, subsample=0.8, random\_state=42)),**

**('rf', RandomForestRegressor(n\_estimators=200, max\_depth=10, random\_state=42)),**

**('xgb', XGBRegressor(learning\_rate=0.05, max\_depth=5,**

**n\_estimators=300, subsample=0.8, random\_state=42))**

**],**

**final\_estimator=LinearRegression()**

**)**

**stacking.fit(X\_train\_scaled, y\_train)**

**stacking\_pred = stacking.predict(X\_test\_scaled)**

**stacking\_mae = mean\_absolute\_error(y\_test, stacking\_pred)**

**stacking\_r2 = r2\_score(y\_test, stacking\_pred)**

**print(f"Stacking Regressor: MAE = {stacking\_mae:.4f}, R² = {stacking\_r2:.4f}")**

**# Save ensemble results**

**ensemble\_results = pd.DataFrame({**

**'Model': ['Simple Averaging', 'Voting Regressor', 'Stacking Regressor'],**

**'MAE': [avg\_mae, voting\_mae, stacking\_mae],**

**'R2': [avg\_r2, voting\_r2, stacking\_r2]**

**})**

**print("\nEnsemble Performance Summary:")**

**print(ensemble\_results)**

**# Save ensemble models**

**joblib.dump(voting, 'voting\_ensemble.joblib')**

**joblib.dump(stacking, 'stacking\_ensemble.joblib')**

**# ======================**

**# UPDATE PREDICTIONS WITH ENSEMBLES**

**# ======================**

**def predict\_with\_ensembles(compositions, feature\_labels, imputer, scaler):**

**"""Predict band gaps including ensembles"""**

**# First get individual model predictions**

**predictions = predict\_bandgaps(compositions, feature\_labels, imputer, scaler, models)**

**# Generate features for new compositions**

**pred\_df = pd.DataFrame({'composition': compositions})**

**pred\_df['comp\_obj'] = pred\_df['composition'].apply(Composition)**

**X\_pred = featurizer.featurize\_dataframe(pred\_df, 'comp\_obj', ignore\_errors=True)**

**X\_pred = meredig\_featurizer.featurize\_dataframe(X\_pred, 'comp\_obj', ignore\_errors=True)**

**X\_pred = oxidation\_featurizer.featurize\_dataframe(X\_pred, 'comp\_obj', ignore\_errors=True)**

**X\_pred, \_ = add\_periodic\_features(X\_pred, periodic\_table)**

**X\_pred = X\_pred.reindex(columns=feature\_labels, fill\_value=np.nan)**

**X\_pred\_imputed = imputer.transform(X\_pred)**

**X\_pred\_scaled = scaler.transform(X\_pred\_imputed)**

**# Add traditional ensemble predictions**

**predictions['Simple Averaging'] = np.round(**

**(predictions['LightGBM'] + predictions['Random Forest'] + predictions['XGBoost']) / 3, 3**

**)**

**predictions['Voting'] = np.round(voting.predict(X\_pred\_scaled), 3)**

**predictions['Stacking'] = np.round(stacking.predict(X\_pred\_scaled), 3)**

**return predictions**

**# Get updated predictions**

**print("\nGenerating predictions with ensembles...")**

**final\_predictions = predict\_with\_ensembles(**

**target\_compositions,**

**feature\_labels,**

**imputer,**

**scaler**

**)**

**# Save final results**

**final\_predictions.to\_csv("final\_predictions\_with\_ensembles1706for5composition.csv", index=False)**

**print("\nFinal Predictions with Ensembles:")**

**print(final\_predictions)**