```python
# Install required packages
# pip install xgboost lightgbm scikit-learn pandas numpy matplotlib seaborn

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.impute import SimpleImputer
from sklearn.feature_selection import SelectKBest, f_classif, mutual_info_classif
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                           f1_score, confusion_matrix, mean_squared_error,
                           mean_absolute_error, classification_report)
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, GradientBoostingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression, LinearRegression, Lasso
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
import xgboost as xgb
import lightgbm as lgb
import warnings
warnings.filterwarnings('ignore')



class DataPreprocessor:
    def __init__(self):
        self.scaler = MinMaxScaler()
        self.imputer_median = SimpleImputer(strategy='median')
        self.imputer_mode = SimpleImputer(strategy='most_frequent')
        self.label_encoders = {}

    def preprocess_data(self, df):
        """Complete preprocessing pipeline"""
        # Drop irrelevant columns
        columns_to_drop = ['Timestamp', 'comments', 'state']
        df_processed = df.drop(columns=[col for col in columns_to_drop if col in df.columns])

        # Handle missing values
        df_processed = self._handle_missing_values(df_processed)

        # Encode categorical variables
        df_processed = self._encode_categorical(df_processed)

        return df_processed

    def _handle_missing_values(self, df):
        """Handle missing values using median and mode imputation"""
        numeric_cols = df.select_dtypes(include=[np.number]).columns
        categorical_cols = df.select_dtypes(include=['object']).columns

        # Median imputation for numeric columns
        if len(numeric_cols) > 0:
            df[numeric_cols] = self.imputer_median.fit_transform(df[numeric_cols])

        # Mode imputation for categorical columns
        if len(categorical_cols) > 0:
            df[categorical_cols] = self.imputer_mode.fit_transform(df[categorical_cols])

        return df

    def _encode_categorical(self, df):
        """Encode categorical variables using Label Encoding"""
        categorical_cols = df.select_dtypes(include=['object']).columns

        for col in categorical_cols:
            le = LabelEncoder()
            df[col] = le.fit_transform(df[col].astype(str))
            self.label_encoders[col] = le

        return df

    def normalize_features(self, X_train, X_test=None):
        """Apply Min-Max normalization"""
        X_train_scaled = self.scaler.fit_transform(X_train)
```

```
        if X_test is not None:
            X_test_scaled = self.scaler.transform(X_test)
            return X_train_scaled, X_test_scaled

        return X_train_scaled



class EnhancedCoatiOptimization:
    def __init__(self, n_features=10, max_iter=50):
        self.n_features = n_features
        self.max_iter = max_iter
        self.selected_features = None

    def _opposition_based_learning(self, population):
        """Implement Opposition-Based Learning (OBL)"""
        opposite_pop = []
        for individual in population:
            opposite = 1 - individual  # For binary representation
            opposite_pop.append(opposite)
        return np.array(opposite_pop)

    def _fitness_function(self, features, X, y):
        """Fitness function based on feature importance"""
        if np.sum(features) == 0:
            return 0

        selected_idx = np.where(features == 1)[0]
        if len(selected_idx) == 0:
            return 0

        X_selected = X[:, selected_idx]

        # Use mutual information as fitness measure
        mi_scores = mutual_info_classif(X_selected, y)
        return np.mean(mi_scores)

    def fit(self, X, y):
        """Fit the Enhanced Coati Optimization algorithm"""
        n_samples, n_total_features = X.shape
        pop_size = min(20, n_total_features)

        # Initialize population (binary representation)
        population = np.random.randint(0, 2, (pop_size, n_total_features))

        # Apply Opposition-Based Learning
        opposite_pop = self._opposition_based_learning(population)
        extended_pop = np.vstack([population, opposite_pop])

        # Evaluate fitness and select best individuals
        fitness_scores = []
        for individual in extended_pop:
            fitness = self._fitness_function(individual, X, y)
            fitness_scores.append(fitness)

        # Select top individuals
        best_indices = np.argsort(fitness_scores)[-pop_size:]
        population = extended_pop[best_indices]

        # Evolution process (simplified)
        for iteration in range(self.max_iter):
            new_population = []

            for i in range(pop_size):
                # Coati behavior simulation (simplified)
                if np.random.random() < 0.5:
                    # Exploration phase
                    new_individual = np.random.randint(0, 2, n_total_features)
                else:
                    # Exploitation phase
                    best_idx = np.argmax([self._fitness_function(ind, X, y) for ind in population])
                    new_individual = population[best_idx].copy()

                    # Random mutation
                    mutation_prob = 0.1
                    for j in range(n_total_features):
                        if np.random.random() < mutation_prob:
```

```python
                        new_individual[j] = 1 - new_individual[j]

                new_population.append(new_individual)

            population = np.array(new_population)

        # Select best solution
        final_fitness = [self._fitness_function(ind, X, y) for ind in population]
        best_solution = population[np.argmax(final_fitness)]

        # Ensure we select exactly n_features
        if np.sum(best_solution) > self.n_features:
            selected_indices = np.where(best_solution == 1)[0]
            # Sort by importance and select top n_features
            importances = []
            for idx in selected_indices:
                temp_features = np.zeros(n_total_features)
                temp_features[idx] = 1
                importances.append(self._fitness_function(temp_features, X, y))

            top_indices = selected_indices[np.argsort(importances)[-self.n_features:]]
            best_solution = np.zeros(n_total_features)
            best_solution[top_indices] = 1

        self.selected_features = np.where(best_solution == 1)[0]
        return self

    def transform(self, X):
        """Transform data using selected features"""
        return X[:, self.selected_features]

    def fit_transform(self, X, y):
        """Fit and transform data"""
        self.fit(X, y)
        return self.transform(X)



class WhiteSharkOptimization:
    def __init__(self, n_sharks=20, max_iter=100):
        self.n_sharks = n_sharks
        self.max_iter = max_iter
        self.best_params = None

    def optimize_xgboost_params(self, X, y):
        """Optimize XGBoost parameters using White Shark Optimization"""
        # Parameter bounds for XGBoost
        param_bounds = {
            'max_depth': (3, 10),
            'learning_rate': (0.01, 0.3),
            'n_estimators': (50, 300),
            'subsample': (0.6, 1.0),
            'colsample_bytree': (0.6, 1.0)
        }

        # Initialize shark population
        population = self._initialize_population(param_bounds)

        best_fitness = -np.inf
        best_solution = None

        for iteration in range(self.max_iter):
            for i in range(self.n_sharks):
                # Evaluate fitness (cross-validation score)
                params = self._decode_solution(population[i], param_bounds)
                fitness = self._evaluate_fitness(params, X, y)

                if fitness > best_fitness:
                    best_fitness = fitness
                    best_solution = population[i].copy()

                # Update shark position (simplified WSO)
                if np.random.random() < 0.5:
                    # Hunting behavior
                    population[i] = self._hunting_behavior(population[i], best_solution)
                else:
                    # Random exploration
```

```python
                    population[i] = self._random_exploration(population[i], param_bounds)

        self.best_params = self._decode_solution(best_solution, param_bounds)
        return self.best_params

    def _initialize_population(self, param_bounds):
        """Initialize shark population"""
        population = []
        for _ in range(self.n_sharks):
            shark = np.random.random(len(param_bounds))
            population.append(shark)
        return np.array(population)

    def _decode_solution(self, solution, param_bounds):
        """Decode normalized solution to actual parameters"""
        params = {}
        param_names = list(param_bounds.keys())

        for i, param_name in enumerate(param_names):
            min_val, max_val = param_bounds[param_name]
            if param_name in ['max_depth', 'n_estimators']:
                params[param_name] = int(min_val + solution[i] * (max_val - min_val))
            else:
                params[param_name] = min_val + solution[i] * (max_val - min_val)

        return params

    def _evaluate_fitness(self, params, X, y):
        """Evaluate fitness using cross-validation"""
        try:
            model = xgb.XGBClassifier(**params, random_state=42)
            scores = cross_val_score(model, X, y, cv=3, scoring='accuracy')
            return np.mean(scores)
        except:
            return -1  # Return low fitness for invalid parameters

    def _hunting_behavior(self, shark, best_shark):
        """Simulate hunting behavior"""
        return shark + np.random.random(len(shark)) * (best_shark - shark)

    def _random_exploration(self, shark, param_bounds):
        """Random exploration"""
        return np.clip(shark + np.random.normal(0, 0.1, len(shark)), 0, 1)



class WSExGBClassifier:
    def __init__(self):
        self.wso = WhiteSharkOptimization()
        self.eco = EnhancedCoatiOptimization()
        self.model = None
        self.best_params = None

    def fit(self, X, y):
        """Fit WS_ExGB model"""
        print("Starting Enhanced Coati Optimization for feature selection...")
        X_selected = self.eco.fit_transform(X, y)
        print(f"Selected {X_selected.shape[1]} features out of {X.shape[1]}")

        print("Starting White Shark Optimization for hyperparameter tuning...")
        self.best_params = self.wso.optimize_xgboost_params(X_selected, y)
        print(f"Optimized parameters: {self.best_params}")

        # Train final model
        self.model = xgb.XGBClassifier(**self.best_params, random_state=42)
        self.model.fit(X_selected, y)

        return self

    def predict(self, X):
        """Make predictions"""
        X_selected = self.eco.transform(X)
        return self.model.predict(X_selected)

    def predict_proba(self, X):
        """Predict probabilities"""
        X_selected = self.eco.transform(X)
```

```
        return self.model.predict_proba(X_selected)


class ModelComparison:
    def __init__(self):
        self.models = {}
        self.results = {}

    def add_models(self, X, y):
        """Add all comparison models"""
        # Proposed model
        self.models['WS_ExGB'] = WSExGBClassifier()

        # Traditional models
        self.models['Random Forest'] = RandomForestClassifier(n_estimators=100, random_state=42)
        self.models['Decision Tree'] = DecisionTreeClassifier(random_state=42)
        self.models['Logistic Regression'] = LogisticRegression(max_iter=1000, random_state=42)
        self.models['AdaBoost'] = AdaBoostClassifier(random_state=42)
        self.models['SVM'] = SVC(probability=True, random_state=42)
        self.models['MLP'] = MLPClassifier(hidden_layer_sizes=(100,), max_iter=500, random_state=42)
        self.models['Gradient Boosting'] = GradientBoostingClassifier(random_state=42)
        self.models['Naive Bayes'] = GaussianNB()
        self.models['XGBoost'] = xgb.XGBClassifier(random_state=42)
        self.models['LightGBM'] = lgb.LGBMClassifier(random_state=42, verbose=-1)

    def evaluate_models(self, X_train, X_test, y_train, y_test):
        """Evaluate all models"""
        for name, model in self.models.items():
            print(f"Training {name}...")

            try:
                # Fit model
                model.fit(X_train, y_train)

                # Make predictions
                y_pred = model.predict(X_test)
                y_pred_proba = None

                if hasattr(model, 'predict_proba'):
                    y_pred_proba = model.predict_proba(X_test)[:, 1]

                # Calculate metrics
                self.results[name] = self._calculate_metrics(y_test, y_pred, y_pred_proba)

            except Exception as e:
                print(f"Error training {name}: {str(e)}")
                self.results[name] = None

    def _calculate_metrics(self, y_true, y_pred, y_pred_proba=None):
        """Calculate all performance metrics"""
        # Classification metrics
        accuracy = accuracy_score(y_true, y_pred)
        precision = precision_score(y_true, y_pred, average='weighted')
        recall = recall_score(y_true, y_pred, average='weighted')
        f1 = f1_score(y_true, y_pred, average='weighted')

        # Confusion matrix for specificity
        cm = confusion_matrix(y_true, y_pred)
        if cm.shape == (2, 2):
            tn, fp, fn, tp = cm.ravel()
            specificity = tn / (tn + fp) if (tn + fp) > 0 else 0
        else:
            specificity = 0  # For multiclass, specificity is more complex

        # Regression-style metrics (treating as continuous)
        rmse = np.sqrt(mean_squared_error(y_true, y_pred))
        mae = mean_absolute_error(y_true, y_pred)

        return {
            'accuracy': accuracy,
            'precision': precision,
            'recall': recall,
            'f1_score': f1,
            'specificity': specificity,
            'rmse': rmse,
            'mae': mae
```

```
        }

    def get_results_dataframe(self):
        """Get results as DataFrame"""
        results_data = []

        for model_name, metrics in self.results.items():
            if metrics is not None:
                row = {'Model': model_name}
                row.update(metrics)
                results_data.append(row)

        return pd.DataFrame(results_data)



def main():
    # Load data
    print("Loading mental health survey data...")
    df = pd.read_csv('/content/survey.csv')

    # Preprocessing
    preprocessor = DataPreprocessor()
    df_processed = preprocessor.preprocess_data(df)

    # Define target variable (treatment as example)
    target_col = 'treatment'

    # Prepare features and target
    X = df_processed.drop(columns=[target_col])
    y = df_processed[target_col]

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    # Normalize features
    X_train_scaled, X_test_scaled = preprocessor.normalize_features(X_train, X_test)

    # Initialize model comparison
    comparison = ModelComparison()
    comparison.add_models(X_train_scaled, y_train)

    # Evaluate models
    print("Evaluating all models...")
    comparison.evaluate_models(X_train_scaled, X_test_scaled, y_train, y_test)

    # Get results
    results_df = comparison.get_results_dataframe()

    # Display results
    print("\n" + "="*80)
    print("MODEL COMPARISON RESULTS")
    print("="*80)

    # Sort by accuracy
    results_df_sorted = results_df.sort_values('accuracy', ascending=False)
    print(results_df_sorted.round(4))

    # Highlight best performing model
    best_model = results_df_sorted.iloc[0]
    print(f"\nBest Performing Model: {best_model['Model']}")
    print(f"Accuracy: {best_model['accuracy']:.4f}")
    print(f"Precision: {best_model['precision']:.4f}")
    print(f"Recall: {best_model['recall']:.4f}")
    print(f"F1-Score: {best_model['f1_score']:.4f}")

    return results_df_sorted

# Run the complete analysis
if __name__ == "__main__":
    results = main()
```

```
Loading mental health survey data...
Evaluating all models...
Training WS_ExGB...
```

```
Starting Enhanced Coati Optimization for feature selection...
Selected 5 features out of 23
Starting White Shark Optimization for hyperparameter tuning...
Optimized parameters: {'max_depth': 3, 'learning_rate': np.float64(0.010000003005809749), 'n_estimators': 84, 'subsample': np.float64(0.
Training Random Forest...
Training Decision Tree...
Training Logistic Regression...
Training AdaBoost...
Training SVM...
Training MLP...
Training Gradient Boosting...
Training Naive Bayes...
Training XGBoost...
Training LightGBM...

================================================================================
MODEL COMPARISON RESULTS
================================================================================
                  Model  accuracy  precision  recall  f1_score  specificity  \
0               WS_ExGB    0.7540     0.7559  0.7540    0.7537       0.7903
1         Random Forest    0.7381     0.7407  0.7381    0.7377       0.7823
4              AdaBoost    0.7302     0.7327  0.7302    0.7298       0.7742
7     Gradient Boosting    0.7262     0.7291  0.7262    0.7257       0.7742
9               XGBoost    0.7183     0.7185  0.7183    0.7183       0.7258
6                   MLP    0.7103     0.7108  0.7103    0.7103       0.7258
3   Logistic Regression    0.6984     0.7023  0.6984    0.6975       0.7581
10             LightGBM    0.6944     0.6964  0.6944    0.6941       0.7339
5                   SVM    0.6905     0.6916  0.6905    0.6903       0.7177
8           Naive Bayes    0.6468     0.6871  0.6468    0.6292       0.8710
2         Decision Tree    0.6151     0.6165  0.6151    0.6146       0.6532

      rmse     mae
0   0.4960  0.2460
1   0.5118  0.2619
4   0.5195  0.2698
7   0.5233  0.2738
9   0.5308  0.2817
6   0.5382  0.2897
3   0.5492  0.3016
10  0.5528  0.3056
5   0.5563  0.3095
8   0.5943  0.3532
2   0.6204  0.3849

Best Performing Model: WS_ExGB
Accuracy: 0.7540
Precision: 0.7559
Recall: 0.7540
F1-Score: 0.7537
```