```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split, KFold
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.utils import resample
import matplotlib.pyplot as plt

# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# The provided data
df=pd.read_csv("/content/synthetic_cycling_data_larger.csv")
# Create a larger dataset through bootstrap resampling
df_augmented = resample(df, replace=True, n_samples=200, random_state=42)

# Define features and target
X = df_augmented.drop('energy_expenditure_kcal', axis=1)
y = df_augmented['energy_expenditure_kcal']

# Set up preprocessing for numerical and categorical features
numeric_features = ['speed_kmph', 'distance_km', 'heart_rate_bpm', 'ride_duration_min', 'elevation_gain_m']
categorical_features = ['terrain_type']

# Create and fit preprocessors
numeric_transformer = StandardScaler()
categorical_transformer = OneHotEncoder(handle_unknown='ignore')

# Fit preprocessors on the entire dataset
X_numeric = numeric_transformer.fit_transform(X[numeric_features])
X_categorical = categorical_transformer.fit_transform(X[categorical_features]).toarray()

# Get feature names after one-hot encoding
categorical_feature_names = categorical_transformer.get_feature_names_out(categorical_features)

# Combine processed features
X_processed = np.hstack((X_numeric, X_categorical))

# Feature names for later analysis
feature_names = numeric_features + categorical_feature_names.tolist()

# Implement k-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
r2_scores = []
rmse_scores = []

fold_count = 1
for train_index, test_index in kf.split(X_processed):
    print(f"Training fold {fold_count}...")

    # Split data
    X_train, X_test = X_processed[train_index], X_processed[test_index]
    y_train, y_test = y.iloc[train_index].values, y.iloc[test_index].values

    # Build Keras model
#     model = keras.Sequential([
#         keras.layers.Dense(20, activation='relu', input_shape=(X_train.shape[1],)),
#         keras.layers.Dropout(0.2),  # Add dropout for regularization
#         keras.layers.Dense(10, activation='relu'),
#         keras.layers.Dropout(0.1),
#         keras.layers.Dense(1)  # Output layer (no activation for regression)
#     ])

#     # Compile model
#     model.compile(
#         optimizer=keras.optimizers.Adam(learning_rate=0.001),
#         loss='mean_squared_error'
#     )

#     # Early stopping callback
#     early_stopping = keras.callbacks.EarlyStopping(
#         monitor='val_loss',
#         patience=20,
#         restore_best_weights=True
#     )

#     # Train model
```

```
#     history = model.fit(
#         X_train, y_train,
#         epochs=200,
#         batch_size=16,
#         validation_split=0.2,
#         callbacks=[early_stopping],
#         verbose=0
#     )

#     # Evaluate model
#     y_pred = model.predict(X_test, verbose=0).flatten()

#     # Calculate performance metrics
#     r2 = r2_score(y_test, y_pred)
#     rmse = np.sqrt(mean_squared_error(y_test, y_pred))

#     r2_scores.append(r2)
#     rmse_scores.append(rmse)
#     fold_count += 1

# # Calculate mean metrics across all folds
# mean_r2 = np.mean(r2_scores)
# mean_rmse = np.mean(rmse_scores)

# print(f"Cross-validated R² score: {mean_r2:.4f}")
# print(f"Cross-validated RMSE: {mean_rmse:.4f}")
# print(f"Approximate model accuracy: {max(0, mean_r2) * 100:.2f}%")

# Train final model on all data
final_model = keras.Sequential([
    keras.layers.Dense(20, activation='relu', input_shape=(X_processed.shape[1],)),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation='relu'),
    keras.layers.Dropout(0.1),
    keras.layers.Dense(1)
])

final_model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss='mean_squared_error'
)

# Train the final model
final_model.fit(
    X_processed, y.values,
    epochs=150,
    batch_size=16,
    verbose=0
)

# Prepare original data for evaluation
X_orig_numeric = numeric_transformer.transform(df[numeric_features])
X_orig_categorical = categorical_transformer.transform(df[categorical_features]).toarray()
X_orig_processed = np.hstack((X_orig_numeric, X_orig_categorical))
y_orig = df['energy_expenditure_kcal'].values

# Evaluate on original data
y_pred_orig = final_model.predict(X_orig_processed, verbose=0).flatten()

r2_orig = r2_score(y_orig, y_pred_orig)
rmse_orig = np.sqrt(mean_squared_error(y_orig, y_pred_orig))

print("\nPerformance on original data:")
print(f"R² score: {r2_orig:.4f}")
print(f"RMSE: {rmse_orig:.4f}")
print(f"Model accuracy on original data: {max(0, r2_orig) * 100:.2f}%")

# Visualize predictions vs actual values
plt.figure(figsize=(10, 6))
plt.scatter(y_orig, y_pred_orig, alpha=0.7)
plt.plot([y_orig.min(), y_orig.max()], [y_orig.min(), y_orig.max()], 'k--', lw=2)
plt.xlabel('Actual Energy Expenditure (kcal)')
plt.ylabel('Predicted Energy Expenditure (kcal)')
plt.title('Keras Neural Network Model\nPredictions vs Actual Values')
plt.grid(True)
plt.show()

# Approximate feature importance using permutation importance
def get_feature_importance_permutation(model, X, y, feature_names, n_repeats=10):
    """Calculate feature importance using permutation importance method"""
    baseline_score = r2_score(y, model.predict(X, verbose=0).flatten())
    importances = []
```

```
        for i in range(X.shape[1]):
            scores = []
            for _ in range(n_repeats):
                # Create a copy of the feature matrix
                X_permuted = X.copy()
                # Permute the values of the current feature
                X_permuted[:, i] = np.random.permutation(X_permuted[:, i])
                # Get predictions and calculate the score
                y_pred = model.predict(X_permuted, verbose=0).flatten()
                score = r2_score(y, y_pred)
                # Calculate importance as decrease in performance
                importance = baseline_score - score
                scores.append(importance)

            # Average importance over repeats
            importances.append(np.mean(scores))

        # Create dictionary of feature importances
        feature_importance = dict(zip(feature_names, importances))
        return feature_importance

    # Calculate feature importance
    feature_importance = get_feature_importance_permutation(
        final_model, X_orig_processed, y_orig, feature_names
    )

    # Sort features by importance
    sorted_features = sorted(feature_importance.items(), key=lambda x: x[1], reverse=True)

    # Print feature importances
    print("\nFeature Importances:")
    for feature, importance in sorted_features:
        print(f"{feature}: {importance:.4f}")



    # Save the model
    final_model.save('cycling_energy_keras_model.h5')
    print("\nModel saved as 'cycling_energy_keras_model.h5'")
```

```
Training fold 1...
Training fold 1...
Training fold 1...
Training fold 1...
Training fold 1...
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` arg
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Performance on original data:
R² score: 0.9172
RMSE: 14.7600
Model accuracy on original data: 91.72%
```