SPORT INJURY PREDICTION

---

1. Introduction

The Sports Injury Prediction System aims to predict the risk of injury for athletes based on various features such as age, training hours, previous injuries, BMI, gender, and sport type. The application utilizes machine learning algorithms, specifically Random Forest and K-Nearest Neighbors (KNN), to provide accurate predictions.

---

2. Dataset Overview

The sports_injury1.csv dataset provides comprehensive information about athletes, which is crucial for predicting injury risk. The key features included in the dataset are:

- **Age**: Represents the age of the athlete, which can influence injury susceptibility.
- **Training_Hours_per_Week**: Indicates the number of hours the athlete trains weekly, reflecting their activity level.
- **Previous_Injuries**: Counts the number of injuries the athlete has experienced in the past, serving as an important risk factor.
- **BMI**: Body Mass Index, a measure of body fat based on height and weight, which can impact physical performance and injury risk.
- **Gender**: Specifies the athlete's gender (Male/Female), which may correlate with injury patterns.
- **Sport**: Identifies the type of sport the athlete participates in (e.g., Basketball, Soccer), as different sports have varying injury risks.
- **Injury_Risk**: The target variable that indicates the likelihood of injury, categorized into risk levels (e.g., Low, Medium, High).

Data Characteristics

- Total records: [Insert number]
- Total features: [Insert number]

---

3. Data Preprocessing

The notebook begins by importing key Python libraries that are fundamental for data analysis and visualization.

```
import pandas as pd
```

```
import numpy as np
import matplotlib.pyplot as plt
from pylab import rcParams
```

Loading the Data

The dataset is loaded using Pandas, and initial checks are performed to understand its structure and identify missing values.

```
desu = pd.read_csv('sports_injury1.csv')
```

Handling Missing Values

Missing values are checked and handled appropriately to ensure data integrity.

```
print(desu.isnull().sum())
```

Data Cleaning

Categorical features are standardized for consistent encoding.

data['Gender'] = data['Gender'].astype(str).str.lower().str.strip()

---

4. Exploratory Data Analysis (EDA)

Descriptive Statistics

The dataset is summarized to understand distributions and key statistics.

The describe() method in Pandas is used to generate a summary of key statistics for the numerical features in the dataset. This summary provides insights into the distribution and characteristics of the data

```
desu.describe()
```

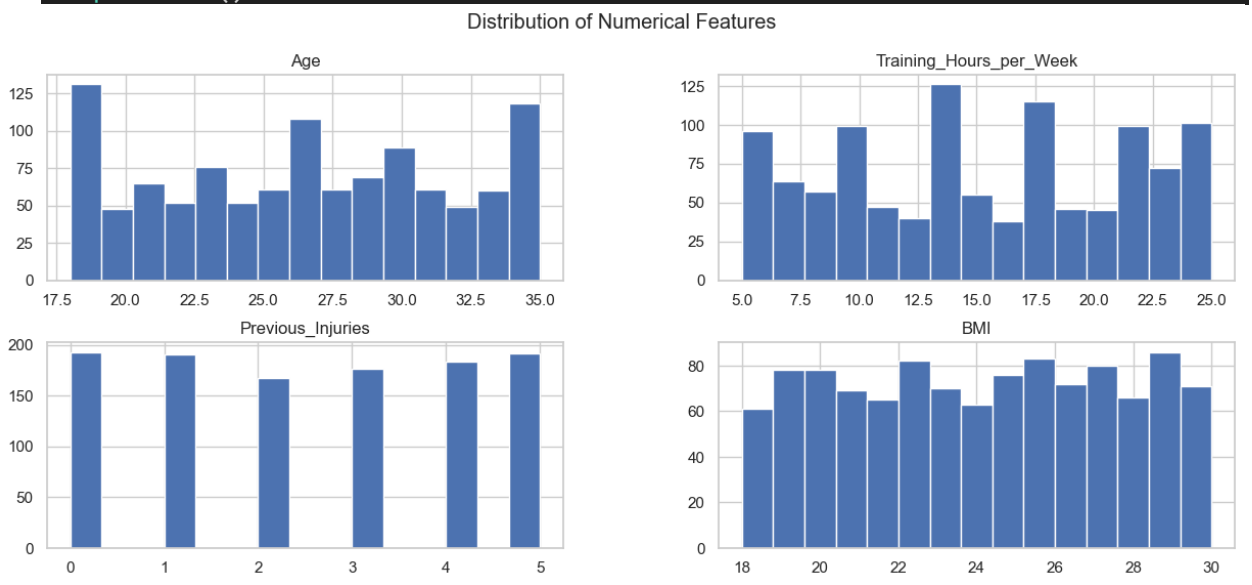
Visualizations

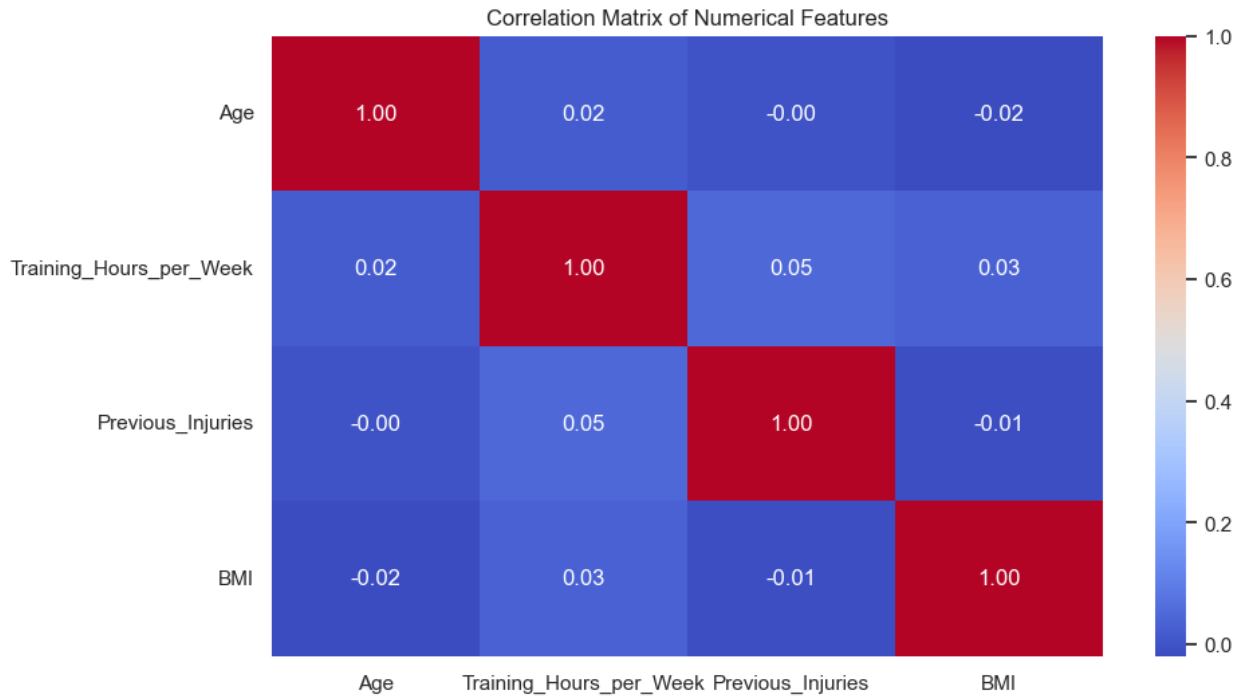
Various plots are generated to visualize distributions and correlations among numerical features.

- **Histograms**: For numerical features.
- **Heatmap**: To show correlation between features.
- **Count Plots**: For categorical features against injury risk.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Set the aesthetic style of the plots
```

```
sns.set(style="whitegrid")

# Distribution of numerical features
numerical_features = ['Age', 'Training_Hours_per_Week',
'Previous_Injuries', 'BMI']
desu[numerical_features].hist(bins=15, figsize=(15, 6), layout=(2, 2))
plt.suptitle('Distribution of Numerical Features')
plt.show()

# Correlation matrix
plt.figure(figsize=(10, 6))
sns.heatmap(desu[numerical_features].corr(), annot=True, cmap='coolwarm',
fmt='.2f')
plt.title('Correlation Matrix of Numerical Features')
plt.show()

# Class distribution of the target variable
plt.figure(figsize=(8, 5))
sns.countplot(x='Injury_Risk', data=desu, palette='Set2')
plt.title('Distribution of Injury Risk')
plt.show()
```





5. Feature Engineering

One-Hot Encoding

One-Hot Encoding is a technique used to convert categorical variables into a numerical format that can be used by machine learning algorithms. This method is particularly useful for handling categorical data, which cannot be directly processed by most algorithms.

Categorical features are transformed into numerical format using one-hot encoding.

```
data = pd.get_dummies(data, columns=['Gender', 'Sport'], drop_first=True)
```

Feature Scaling

Feature Scaling is a crucial preprocessing step in machine learning that involves standardizing the range of independent variables or features. This process helps improve the performance and convergence speed of many algorithms, particularly those that rely on distance calculations, such as K-Nearest Neighbors and gradient descent-based methods.

Standardization with StandardScaler

In the provided code snippet, the StandardScaler from the sklearn.preprocessing module is used to standardize the numerical features in the dataset.

Numerical features are standardized using StandardScaler to improve model performance.

```
from sklearn.preprocessing import StandardScaler

# Select numerical features for scaling
```

```
numerical_features = ['Age', 'Training_Hours_per_Week', 'Previous_Injuries',
'BMI']
scaler = StandardScaler()

# Scale the features
desu[numerical_features] = scaler.fit_transform(desu[numerical_features])
```

6. Model Selection

Selected Models for Training

Two machine learning models are chosen for training and predicting injury risk based on the dataset:

1. **Random Forest**

Description: Random Forest is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of their predictions (for classification tasks) or the mean prediction (for regression tasks).

Advantages:

Improved Accuracy: By averaging the results of multiple trees, Random Forest reduces the risk of overfitting and improves prediction accuracy.

Handles Complexity: It effectively captures complex relationships and interactions between features in the data.

Feature Importance: Provides insights into feature importance, helping to identify which variables are most influential in making predictions.

2. **K-Nearest Neighbors (KNN)**

Description: KNN is a simple, instance-based learning algorithm that classifies a data point based on the majority class of its 'k' nearest neighbors in the feature space.

Advantages:

Simplicity: KNN is easy to understand and implement, making it a good choice for beginners.

No Training Phase: The model does not require a training phase; it simply memorizes the training data and makes predictions based on proximity.

Flexibility: Can be used for both classification and regression tasks, adapting to various types of data.

Two models are chosen for training:

- **Random Forest:** An ensemble method that builds multiple decision trees to improve prediction accuracy and handle complex relationships in the data.

K-Nearest Neighbors (KNN): A straightforward algorithm that classifies data points based on the proximity to other data points in the feature space.

---

7. Model Training

Model training involves preparing the machine learning models to make predictions based on the dataset. The process typically includes splitting the data into training and testing sets, followed by fitting the models on the training data.

Data Splitting

In the provided code snippet, the dataset is split into training and testing sets using the train_test_split function:

The dataset is split into training and testing sets, and models are trained using the training data.

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

X: Represents the feature set (independent variables).

y: Represents the target variable (dependent variable).

test_size=0.2: Specifies that 20% of the data will be allocated for testing, while 80% will be used for training.

random_state=42: Ensures reproducibility of the results by setting a seed for random number generation.

stratify=y: Ensures that the split maintains the same proportion of classes in both training and testing sets, which is particularly important for imbalanced datasets.

Once the data is split, the models (e.g., Random Forest and KNN) are trained using the training data (X_train and y_train). During this phase, the models learn the relationships between the features and the target variable, allowing them to make predictions on unseen data.

8. Model Evaluation

Model evaluation is a crucial step in the machine learning process, allowing us to assess how well our models perform on unseen data. Here's a brief overview of the evaluation methods used for both the Random Forest and K-Nearest Neighbors (KNN) models.

Accuracy and Classification Report

For both models, the evaluation includes calculating accuracy scores and generating classification reports.

The performance of each model is evaluated using accuracy scores and classification reports.

Random Forest Evaluation

```
print("Random Forest Evaluation:")
print(confusion_matrix(y_test, y_pred_rf))
print(classification_report(y_test, y_pred_rf))
print(f"Random Forest Accuracy: {accuracy_score(y_test, y_pred_rf):.2f}")
```

Confusion Matrix

Confusion Matrix: This matrix displays the counts of true positive, true negative, false positive, and false negative predictions. It helps visualize how well the model is performing in distinguishing between classes.

Confusion matrices are generated to visualize model performance.

Classification Report: This report provides detailed metrics, including:

Precision: The ratio of true positive predictions to the total predicted positives.

Recall: The ratio of true positive predictions to the actual positives.

F1 Score: The harmonic mean of precision and recall, providing a balance between the two.

Accuracy Score: This score indicates the proportion of correct predictions made by the model out of all predictions. It is calculated as:

```
# Evaluate the KNN model
print("KNN Evaluation:")
print(confusion_matrix(y_test, y_pred_knn))
print(classification_report(y_test, y_pred_knn))
print(f"KNN Accuracy: {accuracy_score(y_test, y_pred_knn):.2f}")
```

Similar to the Random Forest evaluation, the KNN model's performance is assessed using:

Confusion Matrix: Visualizes the model's predictions against actual labels.

Classification Report: Provides key metrics to evaluate the KNN model's effectiveness in classifying data points.

9. Hyperparameter Tuning

Hyperparameter tuning is a critical step in optimizing machine learning models, where the goal is to identify the best set of hyperparameters that improve model performance.

Hyperparameters are parameters that are not learned from the data but are set before the training process begins. Examples for Random Forest include the number of trees (n_estimators), maximum depth of the trees (max_depth), and minimum samples required to split a node (min_samples_split).

GridSearchCV:

GridSearchCV is a method from the sklearn.model_selection module that performs an exhaustive search over specified hyperparameter values for a given model. It evaluates every combination of hyperparameters using cross-validation to find the optim

Grid search is utilized to find the optimal hyperparameters for the Random Forest model, improving its performance.

```
# Fit GridSearchCV
grid_search.fit(X_train, y_train)
```

10. Deployment

Deployment is the final stage in the machine learning workflow, where trained models are made available for use in real-world applications. This process typically involves saving the models and creating an interface for users to interact with them.

Saving the Models

To ensure that the trained models can be reused for future predictions, they are saved using Joblib, a library that efficiently serializes Python objects. The code snippet demonstrates how this is done:

The trained models are saved using Joblib for future use in predictions.

```
# Save the models using joblib
joblib.dump(rf_model, 'model_rf.joblib')
joblib.dump(knn_model, 'model_knn.joblib')
joblib.dump(label_encoder, 'label_encoder.joblib')
```

rf_model: The trained Random Forest model is saved to a file named model_rf.joblib.

knn_model: The trained K-Nearest Neighbors model is saved to model_knn.joblib.

label_encoder: Any necessary preprocessing components, such as label encoders, are also saved to ensure consistent data handling during predictions.

FastAPI Application

A FastAPI application is then created to serve predictions based on user input. FastAPI is a modern web framework for building APIs with Python, known for its speed and ease of use.

Purpose: The FastAPI application allows users to send requests with input data, which the application processes to generate predictions using the saved models.

Interaction: Users can interact with the models through a RESTful API, making it convenient to integrate the machine learning models into web applications or other services.

Deployment involves saving trained models for future use and creating an accessible interface for users to obtain predictions. By using Joblib for model serialization and FastAPI for serving predictions, the deployment process ensures that the models can be effectively utilized in real-world scenarios, enhancing their practical value.

A FastAPI application is created to serve predictions based on user input.

---

11. Conclusion

The Sports Injury Prediction System effectively predicts injury risks for athletes using advanced machine learning techniques. Through a systematic approach that includes data preprocessing, model training, hyperparameter tuning, and deployment, the system demonstrates its capability to provide valuable insights into injury prevention

Model Performance: Various models, including Random Forest and K-Nearest Neighbors, were evaluated and optimized to achieve high accuracy.

Hyperparameter Tuning: Techniques like Grid Search were employed to fine-tune model parameters, further enhancing performance.

Deployment: The trained models were saved using Joblib and integrated into a FastAPI application, making predictions accessible to users.