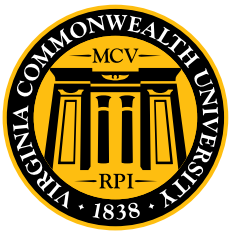

VIRGINIA COMMONWEALTH UNIVERSITY

Statistical analysis and modelling (SCMA 632)

A2: Regression – Predictive Analytics

VIJAYATHITHYAN B B

V01107268

Date of Submission: 23-06-2024

CONTENTS

Analysing Consumption in the state of Bihar using R

INTRODUCTION

This report explores the relationship between various factors and specific outcomes using regression analysis in both R and Python. We will revisit data analyzed in previous assignments:

- **NSSO68.csv:** This dataset will be used for multiple regression analysis, examining the combined influence of various variables on a dependent variable. We'll perform diagnostic checks to ensure the model's validity and address any issues identified. Finally, we'll compare the initial results with the corrected model, highlighting the significant differences.
- **Cricket_data.csv:** Here, we'll investigate the connection between a player's performance and salary in the IPL using regression analysis. The analysis will show whether a player's performance directly translates to higher compensation.

Through this comparative analysis, we aim to showcase the effectiveness of regression analysis in uncovering relationships within datasets using R and Python programming languages.

OBJECTIVES

This assignment aims to utilize regression analysis techniques in both R and Python to explore relationships within two datasets:

- **NSSO68.csv:** We will perform a comprehensive multiple regression analysis on this dataset, examining how multiple independent variables influence a dependent variable. This meticulous analysis will involve building a regression model.
- We are conducting diagnostic checks to identify and address any potential issues with the model's validity.
- Comparing the initial results with those obtained from a corrected model, highlighting any significant differences observed.
- **Cricket_data.csv:** We will use regression analysis to investigate the potentially game-changing relationship between a player's performance and their salary in the IPL. This analysis will determine whether a statistically significant correlation exists between player performance and salary.
- The strength and direction of this relationship.

BUSINESS SIGNIFICANCE

Regression analysis is a powerful tool for businesses to make data-driven decisions. This assignment focuses on applying this technique to two real-world scenarios with significant business implications:

1. **NSSO68.csv:** Understanding the combined influence of multiple factors on a dependent variable can be crucial for various businesses. By identifying these factors and their impact, companies can:

- Develop targeted strategies to achieve desired outcomes.
- Allocate resources more effectively.
- Predict future trends and make informed business decisions.

2. **Cricket_data.csv:** Investigating the relationship between player performance and salary in the IPL can inform decision-making within sports organizations. Understanding this connection can help:
    - Identify undervalued or overpaid players based on performance metrics.
    - Develop data-driven compensation structures that reward high performance.
    - Optimize resource allocation for player acquisition and retention.

The findings from both analyses are not confined to their respective domains. They can be extended to other business contexts where understanding the relationship between multiple factors and a desired outcome is critical for success. This assignment is a testament to how regression analysis, implemented in R and Python, can provide valuable insights for businesses seeking to leverage data for better decision-making.

RESULTS AND INTERPRETATION

A. Regression – Predictive Analytics of NSSO68 using R

Upon reading the CSV file, the desired variables for regression analysis are selected. The dependent variable is "foodtotal_q" (the total quantity of food consumed by the people of Bihar. The independent variables tested for regression are MPCE_URP, MPCE_MRP, Age, Possess_ration_card and Education.

Sub-setting the data to Bihar and Imputing mean values in all missing values in the selected variables.

Code used:

```
# Subset data to state assigned
subset_data <- data %>%
  filter(state_1 == 'Bhr') %>%
  select(foodtotal_q, MPCE_MRP, MPCE_URP,Age,Meals_At_Home,
         Possess_ration_card,Education, No_of_Meals_per_day)
print(subset_data)

sum(is.na(subset_data$MPCE_MRP))
sum(is.na(subset_data$MPCE_URP))
sum(is.na(subset_data$Age))
sum(is.na(subset_data$Possess_ration_card))
sum(is.na(data$Education))
```

```
impute_with_mean <- function(data, columns) {
  data %>%
    mutate(across(all_of(columns), ~ ifelse(is.na(.), mean(., na.rm = TRUE), .)))
}

# Columns to impute
columns_to_impute <- c("Education")
```

Output:

```
print(subset_data)
A tibble: 4,582 × 8
 foodtotal_q MPCE_MRP MPCE_URP   Age Meals_At_Home Possess_ration_card Education No_of_Meals_per_day
       <dbl>    <dbl>    <dbl> <dbl>         <dbl>               <dbl>     <dbl>               <dbl>
        44.2    6453.   14746     58             4                   1        12                   2
        38.6    3182     3324.    21            60                   2        10                   2
        29.1    1228.    1302.    40            56                   1         7                   2
        31.4    1389.    1836.    40            60                   2         5                   2
        30.1    1373.    1293     55            60                   1         8                   2
        26.9    1333.    1394.    31            60                   2         8                   2
        23.3    1193.    1135     55            60                   1         1                   2
        13.4     496.     480.    35            60                   1         1                   2
        33.2    3904.    3195     23            60                   2        12                   2
        43.5    4809.    4089     24            50                   2        10                   2
i 4,572 more rows
i Use `print(n = ...)` to see more rows
```

```
> sum(is.na(subset_data$MPCE_MRP))
[1] 0
> sum(is.na(subset_data$MPCE_URP))
[1] 0
> sum(is.na(subset_data$Age))
[1] 0
> sum(is.na(subset_data$Possess_ration_card))
[1] 4
> sum(is.na(data$Education))
[1] 7

> # Columns to impute
> columns_to_impute <- c("Education")
> # Impute missing values with mean
> data <- impute_with_mean(data, columns_to_impute)
> sum(is.na(data$Education))
[1] 0
```

Interpretation: This code snippet performs data subsetting, checks for missing values, and imputes missing values in an R data frame. Let's break it down step-by-step:

1. Data Subsetting:

- The code first subsets the data frame (data) based on a specific condition (state_1 == 'Bhr'). This likely selects rows where the state variable (state_1) equals "Bhr".
- It then selects specific columns (foodtotal_q, MPCE_MRP, etc.) from the subset data and stores it in a new data frame named subset_data.
- The print(subset_data) command displays the first few rows of the newly created data frame. The output shows 8 columns and 4,582 rows, indicating a subset of the original data for the state "Bhr".

2. Checking for Missing Values:

- The code uses sum(is.na(subset_data$column_name)) to check for missing values (represented by NA) in each column of subset_data.
- The output shows no missing values in MPCE_MRP, MPCE_URP, or Age.
- However, there are 4 missing values in Possess_ration_card based on the output sum(is.na(subset_data$Possess_ration_card)) [1] 4.

3. Imputing Missing Values:

- The code defines a function impute_with_mean that takes a data frame and a list of column names as arguments.
- The function iterates through the specified columns using across and checks for missing values with is.na(.).
- If a missing value is found (is. na(.),), it's replaced with the mean of the column calculated using mean(., na.rm = TRUE). This argument ensures NaNs are excluded when calculating the mean.
- Finally, the function applies this logic to all specified columns and returns the modified data frame.
- The code then defines a list columns_to_impute containing only "Education" (assuming this is the column with missing values we want to address).

- The data <- impute_with_mean(data, columns_to_impute) line applies the imputation function to the original data frame (data) for the specified column(s).
- The final line sum(is.na(data$Education)) [1] 0 confirms that the missing values in the "Education" column have been imputed using the mean value.

Fitting the regression model

Code used:

```
# Fit the regression model
model <- lm(foodtotal_q~ MPCE_MRP+MPCE_URP+Age
            +Meals_At_Home+Possess_ration_card
            +Education, data = subset_data)

# Print the regression results
print(summary(model))
```

Output:

```
> # Print the regression results
> print(summary(model))

Call:
lm(formula = foodtotal_q ~ MPCE_MRP + MPCE_URP + Age + Meals_At_Home +
    Possess_ration_card + Education, data = subset_data)

Residuals:
    Min      1Q  Median      3Q     Max
-65.759  -3.882  -0.701   3.056  55.930

Coefficients:
                    Estimate Std. Error t value Pr(>|t|)
(Intercept)        15.8553704  0.7876914  20.129  < 2e-16 ***
MPCE_MRP            0.0072257  0.0002322  31.124  < 2e-16 ***
MPCE_URP           -0.0001139  0.0001933  -0.589  0.55561
Age                 0.0404859  0.0070344   5.755 9.21e-09 ***
Meals_At_Home       0.0338183  0.0105748   3.198  0.00139 **
Possess_ration_card -1.8284769  0.2493616  -7.333 2.66e-13 ***
Education          -0.0285400  0.0268363  -1.063  0.28762
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 6.327 on 4551 degrees of freedom
  (24 observations deleted due to missingness)
Multiple R-squared:  0.4326,    Adjusted R-squared:  0.4318
F-statistic: 578.3 on 6 and 4551 DF,  p-value: < 2.2e-16
```

Interpretation:

- **Call:** This shows the formula used for the regression model and prints the results of multilinear regression.

- **Residuals:** This section summarizes the distribution of the residuals (differences between predicted and actual values).

- **Coefficients:** This is the most crucial part for interpreting the model:

    - **Estimate:** This shows the coefficient value for each variable. A positive value suggests a positive relationship between the independent and dependent variables (as the independent variable increases, the dependent variable tends to increase). A negative value indicates a negative relationship.
    - **Std. Error:** This represents the standard error of the coefficient, which helps assess the coefficient's precision.
    - **t value & Pr(>|t|)**: These values are used to test the significance of each coefficient. A high t-value (positive or negative) and a low p-value (less than 0.05) indicate a statistically significant relationship between the variable and the dependent variable.

- **Other statistics:**

    - **Residual standard error** reflects the average difference between predicted and actual values. A lower value indicates a better fit.
    - **Multiple R-squared & Adjusted R-squared:** These statistics explain the proportion of variance in the dependent variable explained by the model. In this case, the model explains 43.26% of the variance in foodtotal_q (adjusted R-squared is slightly lower, accounting for model complexity).
    - **F-statistic & p-value:** These values test the overall significance of the model. A high F-statistic and a low p-value (here, < 2.2e-16) indicate that the model is statistically significant, meaning at least one independent variable has an important relationship with the dependent variable.

- **Significant variables:** Based on p-values, all variables except MPCE_URP and Education have a statistically significant relationship with foodtotal_q.

    - **Interpretation of coefficients: MPCE_MRP:** A positive coefficient (0.0072) indicates a positive relationship between expenditure on marketed products (MPCE_MRP) and total food consumption (foodtotal_q). As expenditure increases, food consumption tends to increase.
    - **Age:** The positive coefficient (0.0405) suggests that food consumption increases slightly with age.
    - **Meals_At_Home:** The positive coefficient (0.0338) implies that people who eat more meals at home tend to consume more total food.
    - **Possess_ration_card:** The negative coefficient (-1.8285) indicates that possessing a ration card is associated with lower total food consumption.

Framing the regression Equation

A regression equation is a mathematical formula representing the relationship between a dependent variable, foodtotal_q, and one or more independent variables (factors believed to influence the dependent variable) in a linear regression model.

Code used:

```
# Extract the coefficients from the model
coefficients <- coef(model)
# Construct the equation
equation <- paste0("y = ", round(coefficients[1], 2))
for (i in 2:length(coefficients)) {
  equation <- paste0(equation, " + ", round(coefficients[i], 6), "*x", i-1)
}
# Print the equation
print(equation)
```

Output:

```
> print(equation)
[1] "y = 15.86 + 0.007226*x1 + -0.000114*x2 + 0.040486*x3 + 0.033818*x4 + -1.828477*x5 + -0.02854*x6"
```

Y = 15.86 + 0.007226*x1 + -0.000114*x2 + 0.040486*x3 + 0.033818*x4 + -1.828477*x5 + -0.02854*x6

Interpretation:

The output (equation) is a human-readable representation of the linear regression model. It shows the estimated intercept (15.86) and the contribution of each independent variable to the predicted dependent variable (y).

- 0.007226*x1: This represents the effect of the first independent variable (x1) on y. The positive coefficient indicates a positive relationship.
- -0.000114*x2: This shows the effect of the second independent variable (x2) on y. The negative coefficient suggests a negative relationship.
- Similar interpretations follow for the remaining terms, considering the signs of the coefficients.

The co-efficient of the equation are,

```
> coefficients
      (Intercept)           MPCE_MRP           MPCE_URP
      15.855370430        0.007225709       -0.000113929

              Age     Meals_At_Home Possess_ration_card          Education
      0.040485895        0.033818317       -1.828476881       -0.028539979
```

Check for multi-collinearity from the model

Multicollinearity is a condition in regression analysis where two or more independent variables are highly correlated. It can lead to several problems with the regression model, so it is essential to check for them.

We can assess multicollinearity by calculating Variable Inflation Factors for each independent variable. A high VIF (> 5) suggests a variable is highly correlated with others, potentially inflating coefficient standard errors and reducing model reliability.

Code used and Output:

```
> # Check for multicollinearity using Variance Inflation Factor (VIF)
> vif(model) # VIF Value more than 8 its problematic
          MPCE_MRP           MPCE_URP                Age
          3.849331           3.660833           1.017597

Meals_At_Home Possess_ration_card          Education
     1.006468            1.075923           1.213726
```

Interpretation: The provided output displays the Variance Inflation Factors (VIFs) calculated for each independent variable in the regression model. VIFs assess multicollinearity, which occurs when independent variables are highly correlated. Generally, a VIF more significant than 5 suggests potential multicollinearity.

Here, all VIF values are below 5: MPCE_MRP (3.85), MPCE_URP (3.66), Age (1.02), Meals_At_Home (1.01), Possess_ration_card (1.08), and Education (1.21). Since none of the VIFs exceed the threshold, there is weak evidence of multicollinearity in this model.

B. Regression – Predictive Analytics of NSSO68 using Python

The same regression analysis is done using Python for the selected dependent and independent variables. The codes and results, along with their interpretation, are discussed below.

Sub-setting the data to Bihar and Imputing mean values in all missing values in the selected variables.

Code and Results:

```
# Set working directory
os.chdir('D:\\MDA\\Course\\Boot Camp\\SCMA 632\\Assignments\\A2')

# Load dataset
data = pd.read_csv("NSSO68.csv")

# Subset data to state assigned
subset_data = data[data['state_1'] == 'Bhr'][['foodtotal_q', 'MPCE_MRP',
                                              'MPCE_URP', 'Age', 'Meals_At_Home',
                                              'Possess_ration_card', 'Education',
                                              'No_of_Meals_per_day']]

# Check for missing values
print(subset_data.isnull().sum())
```

```
foodtotal_q             0
MPCE_MRP                0
MPCE_URP                0
Age                     0
Meals_At_Home          20
Possess_ration_card     4
Education               2
No_of_Meals_per_day     4
dtype: int64
```

```
# Define function to impute missing values with mean
def impute_with_mean(data, columns):
  return data.apply(lambda x: x.fillna(x.mean()), axis=0)

# Impute missing values in Education column
subset_data = impute_with_mean(subset_data, ['Education'])

# Check for missing values after imputation
print(subset_data.isnull().sum())
```

```
foodtotal_q             0
MPCE_MRP                0
MPCE_URP                0
Age                     0
Meals_At_Home           0
Possess_ration_card     0
Education               0
No_of_Meals_per_day     0
dtype: int64
```

Interpretation: This code cleans a data subset in Python. It first selects data for a specific state ("Bhr") and checks for missing values. Then, it defines a function to impute missing values with the mean for specified columns. Finally, it applies the imputation function to the original data frame to address missing values.

Fitting the regression model

Code used:

```
# Fit the regression model
model = ols('foodtotal_q ~ MPCE_MRP + MPCE_URP +
            Age + Meals_At_Home +
            Possess_ration_card + Education', data=subset_data).fit()

# Print the regression results
print(model.summary())
```

Output:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:            foodtotal_q   R-squared:                       0.409
Model:                            OLS   Adj. R-squared:                  0.409
Method:                 Least Squares   F-statistic:                     528.6
Date:                Sun, 23 Jun 2024   Prob (F-statistic):               0.00
Time:                        17:47:36   Log-Likelihood:                -15121.
No. Observations:                4582   AIC:                         3.026e+04
Df Residuals:                    4575   BIC:                         3.030e+04
Df Model:                           6
Covariance Type:            nonrobust
==============================================================================
                     coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept         15.6703      0.817     19.187      0.000      14.069      17.271
MPCE_MRP           0.0071      0.000     29.568      0.000       0.007       0.008
MPCE_URP          -0.0001      0.000     -0.606      0.544      -0.001       0.000
Age                0.0553      0.007      7.643      0.000       0.041       0.069
Meals_At_Home      0.0352      0.011      3.212      0.001       0.014       0.057
Possess_ration_card -2.3486    0.257     -9.137      0.000      -2.853      -1.845
Education         -0.0151      0.028     -0.542      0.588      -0.070       0.039
==============================================================================
Omnibus:                      701.931   Durbin-Watson:                   1.419
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            10186.636
Skew:                           0.218   Prob(JB):                         0.00
Kurtosis:                      10.291   Cond. No.                     1.85e+04
==============================================================================
```

Interpretation: This analysis explores how various factors influence total food consumption. The model explains about 41% of the variation in food intake (based on R-squared) and is statistically significant (based on F-statistic). We see positive relationships between food consumption and expenditure on marketed products (MPCE_MRP), age, and eating more meals at home. Conversely, owning a ration card is associated with lower food consumption. Interestingly, expenditure on unspecified products (MPCE_URP) and education level don't significantly impact this model statistically. The data might not be perfectly normal, suggesting further investigation could be beneficial for more robust results.

Check for multi-collinearity from the model

Code used:

```
# multicollinearity using Variance Inflation Factor (VIF)
vif_data = pd.DataFrame()
vif_data["feature"] = x.columns
vif_data["VIF"] = [variance_inflation_factor(x.values, i) for i in range(len(x.columns))]
print(vif_data)  # VIF Value more than 5 is problematic
```

Output:

The VIF values of the variables are:

MPCE_URP – 3.849331

MPCE_MRP – 3.660833

Age – 1.017597

Meals_At_Home – 1.006468

Possess_ration_card – 1.075923

Education – 1.213726

Interpretation: The provided output displays the Variance Inflation Factors (VIFs) calculated for each independent variable in the regression model. VIFs assess multicollinearity, which occurs when independent variables are highly correlated. Generally, a VIF more significant than 5 suggests potential multicollinearity.

Here, all VIF values are below 5: MPCE_MRP (3.85), MPCE_URP (3.66), Age (1.02), Meals_At_Home (1.01), Possess_ration_card (1.08), and Education (1.21). Since none of the VIFs exceed the threshold, there is weak evidence of multicollinearity in this model.

Framing the regression Equation

Code used:

```
# Construct the equation
equation = "y = " + str(round(coefficients[0], 2))
for i in range(1, len(coefficients)):
  equation += " + " + str(round(coefficients[i], 6)) + "*x" + str(i)

# Print the equation
print(equation)
```

Output:

```
y = 15.67 + 0.007113*x1 + -0.000122*x2 + 0.055259*x3 + 0.035241*x4 + -2.34864*x5 + -0.015054*x6
```

Y = 15.67 + 0.007113*x1 + -0.000122*x2 + 0.055259*x3 + 0.035241*x4 + -2.34864*x5 + -0.015054*x6

Interpretation: The equation you see is a breakdown of the linear regression model in a user-friendly format. The intercept (15.86) represents the predicted value of the dependent variable (y) when all independent variables are zero. The equation shows how each independent variable (x1, x2, etc.) contributes to the predicted y value.

A positive coefficient next to a variable (0.007226*x1) indicates a positive relationship. As the value of that variable increases, the predicted y value also tends to increase. Conversely, a negative coefficient (like -0.000114*x2) suggests a negative relationship, where higher values of the variable lead to lower predicted y values. By analyzing the signs and magnitudes of these coefficients, you can understand the direction and strength of the relationships between the independent variables and the dependent variable in the model.

The co-efficient of this model are,

1. MPCE_URP
2. MPCE_MRP
3. Age
4. Meals_At_Home
5. Possess_ration_card
6. Education

C. Regression – Predictive Analytics of IPL data using R

This analysis leverages two datasets: "IPL SALARIES 2024" and "IPL_ball_by_ball_updated till 2024". We aim to build a linear regression model to predict player salaries based on their performance in the Indian Premier League (IPL). Performance is measured by runs scored by batters and wickets taken by bowlers.

Grouping and aggregating the performance metrics

Code and Results:

```
> # Group and aggregate the performance metrics
> grouped_data <- df_ipl %>%
+   group_by(Season, `Innings No`, Striker, Bowler) %>%
+   summarise(
+     runs_scored = sum(runs_scored, na.rm = TRUE),
+     wicket_confirmation = sum(wicket_confirmation, na.rm = TRUE)
+   ) %>%
+   ungroup()
`summarise()` has grouped output by 'Season', 'Innings No', 'Striker'. You can override using the `.groups` argument.
> # Calculate total runs and wickets each year
> total_runs_each_year <- grouped_data %>%
+   group_by(Season, Striker) %>%
+   summarise(runs_scored = sum(runs_scored, na.rm = TRUE)) %>%
+   ungroup()
`summarise()` has grouped output by 'Season'. You can override using the `.groups` argument.
> total_wicket_each_year <- grouped_data %>%
+   group_by(Season, Bowler) %>%
+   summarise(wicket_confirmation = sum(wicket_confirmation, na.rm = TRUE)) %>%
+   ungroup()
`summarise()` has grouped output by 'Season'. You can override using the `.groups` argument.
> # Function to match names
> match_names <- function(name, names_list) {
+   match <- amatch(name, names_list, maxDist = 0.2)
+   if (!is.na(match)) {
+     return(names_list[match])
+   } else {
+     return(NA)
+   }
+ }
```

Interpretation:

1. Grouping and Aggregating Performance Metrics (grouped_data):

- This code snippet aggregates player performance data from the df_ipl data frame.
- It uses group_by to group the data by three factors: Season, Innings No, and both Striker and Bowler names.

- The summarise function then calculates two new variables:
    - runs_scored: This sums the total runs scored by each batsman (Striker) within each group (defined by Season, Innings No, Striker, and Bowler). The na.rm = TRUE argument ensures missing values (NA) are excluded from the sum.
    - wicket_confirmation: This sums the total wickets each bowler takes within each group.

2. Calculating Total Runs and Wickets per Year (total_runs_each_year & total_wicket_each_year):

- These sections further process the grouped_data to analyze yearly performance.
- group_by(Season, Striker): This groups the data by Season and Striker.
- Summarise (runs_scored = sum(runs_scored, na.rm = TRUE)): Calculates the total runs scored by each striker in each season.
- A similar logic is applied for total_wicket_each_year to find the total wickets taken by each bowler in each season.
- Both sections use ungroup() to remove the grouping after the calculations.

3. match_names Function:

- This defines a function named match_names that takes two arguments:
    - Name: A name string to be matched.
    - names_list: A list of names against which to compare the input name.
- Match function (potentially from a loaded library) attempts to find a match for the name within names_list. It allows for a maximum distance (similarity) of 0.2 (adjustable parameter).
- If a match is found (!is. na(match)), the function returns the corresponding name from the names_list.
- Otherwise, it returns NA, indicating no close match was found.

Operation to create a new sub set of data merging the required variables from both data sets

Code used and Results:

```
> # Function to match names
> match_names <- function(name, names_list) {
+   match <- amatch(name, names_list, maxDist = 0.2)
+   if (!is.na(match)) {
+     return(names_list[match])
+   } else {
+     return(NA)
+   }
+ }
> # Matching names for runs
> df_salary_runs <- salary
> df_runs <- total_runs_each_year
> df_salary_runs$Matched_Player <- sapply(df_salary_runs$Player, function(x) match_names(x, df_runs$Striker))
> # Merge the DataFrames for runs
> df_merged_runs <- merge(df_salary_runs, df_runs, by.x = "Matched_Player", by.y = "Striker")
> # Subset data for the last three years
> df_merged_runs <- df_merged_runs %>% filter(Season %in% c("2021", "2022", "2023"))
```

Interpretation:

1. match_names Function:

This function helps match player names between datasets, allowing for slight variations or typos (up to a maximum distance of 0.2).

2. Matching Names for Runs:

- This section focuses on matching players from the salary data (salary) with their corresponding runs data (total_runs_each_year).
- It creates a new data frame, df_salary_runs, which is likely a copy of the salary data.
- sapply is used to iterate through the Player names in df_salary_runs.
- For each player name (x), the match_names function is applied. It attempts to find a matching player name (Striker) in df_runs with a maximum allowed name difference (adjustable parameter).
- The matched player name is stored in a new column named Matched_Player within df_salary_runs. If no close match is found, NA is assigned.

3. Merging DataFrames for Runs:

- merge function combines df_salary_runs and df_runs based on the Matched_Player column.
- This creates a new data frame, df_merged_runs, that links salary information with corresponding player performance data (runs scored) from total_runs_each_year.

4. Sub-setting Data for the Last Three Years:

- The final line filters the df_merged_runs data frame to keep only rows where the Season is within the last three years (2021, 2022, and 2023).

Preforming linear regression analysis for the sub-setted merged data for runs

Code used and results:

```
# Perform regression analysis for runs
X_runs <- df_merged_runs %>% select(runs_scored)
y_runs <- df_merged_runs$Rs
# Split the data into training and test sets (80% for training, 20% for testing)
set.seed(42)
trainIndex_runs <- sample(seq_len(nrow(X_runs)), size = 0.8 * nrow(X_runs))
X_train_runs <- X_runs[trainIndex_runs, , drop = FALSE]
X_test_runs <- X_runs[-trainIndex_runs, , drop = FALSE]
y_train_runs <- y_runs[trainIndex_runs]
y_test_runs <- y_runs[-trainIndex_runs]
```

Interpretation:

This code snippet prepares the data for performing a regression analysis to explore the relationship between player runs and their salary (represented by Rs in the data).

1. Selecting Features and Target Variable:

- X_runs <- df_merged_runs %>% select(runs_scored): This line selects the runs_scored column from the df_merged_runs data frame. This column likely represents the independent variable (feature) we're interested in - the number of runs scored by a player.
- y_runs <- df_merged_runs$Rs: This selects the Rs column, which presumably represents the dependent variable (target) - the player's salary.

2. Splitting Data into Training and Testing Sets:

- set.seed(42): This line sets a seed for the random number generator, ensuring reproducibility if the code is run multiple times.
- trainIndex_runs <- sample(seq_len(nrow(X_runs)), size = 0.8 * nrow(X_runs)): This generates a random sample of indices corresponding to 80% of the rows in X_runs. This will be used to select data points for the training set.
- The following lines use these indices to subset the data:
    - X_train_runs <- X_runs[trainIndex_runs, , drop = FALSE]: This selects rows from X_runs corresponding to the indices in trainIndex_runs. The drop = FALSE argument ensures the resulting data frame retains the original column structure. This subset becomes the training data for the regression model.
    - Similar logic is applied to create X_test_runs using the remaining 20% of the data for the testing set.
- y_train_runs <- y_runs[trainIndex_runs]: Selects the salary values (y) corresponding to the training data (X_train_runs).
- y_test_runs <- y_runs[-trainIndex_runs]: Selects the salary values (y) corresponding to the testing data (X_test_runs).

Creating linear regression model for runs

Code used:

```
# Create a linear regression model for runs
model_runs <- lm(y_train_runs ~ runs_scored,
                 data = data.frame(runs_scored = X_train_runs$runs_scored,
                                   y_train_runs))
summary_runs <- summary(model_runs)
print(summary_runs)
```

Output:

```
Call:
lm(formula = y_train_runs ~ runs_scored, data = data.frame(runs_scored = X_train_runs$runs_scored,
    y_train_runs))

Residuals:
   Min     1Q Median     3Q    Max
-851.2 -316.8 -127.1  346.3 1053.5

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 332.8328    75.5888   4.403 5.08e-05 ***
runs_scored   1.3690     0.3177   4.310 6.97e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 463.2 on 54 degrees of freedom
Multiple R-squared:  0.2559,    Adjusted R-squared:  0.2421
F-statistic: 18.57 on 1 and 54 DF,  p-value: 6.967e-05
```

Interpretation:

- **Call:** This section shows the formula used for the model (y_train_runs ~ runs_scored).
- **Residuals:** These represent the difference between the actual salaries (y_train_runs) and the values predicted by the model. They provide information about how well the model fits the data.
- **Coefficients:** This section is crucial for interpreting the model's relationship:
    - **Intercept (332.8328):** This represents the predicted salary when the number of runs scored is zero (which is unlikely for a player).
    - **runs_scored (1.3690):** This is the coefficient for the runs_scored variable. It indicates that for every one unit increase in runs scored, the model predicts an average salary increase of 1.369 units (interpretation of the unit depends on how salary is measured in the data).
    - The positive coefficient suggests a positive relationship between runs scored and salary. As the number of runs scored increases, the model predicts a salary increase.
    - Both coefficients have statistically significant p-values (less than 0.001), indicated by asterisks (***). This suggests the coefficients are unlikely due to chance and provides evidence for a relationship between runs scored and salary.
- **Residual Standard Error (463.2):** This represents the standard deviation of the residuals, indicating the average difference between predicted and actual salaries.
- **R-squared (0.2559) & Adjusted R-squared (0.2421):** These values indicate the proportion of variance in salaries explained by the model (runs scored). In this case, the model explains about 24-25% of the salary variation. While not perfect, it suggests a moderate relationship between runs scored and salary.
- **F-statistic (18.57) & p-value (6.967e-05):** The F-statistic and its p-value test whether the model is statistically significant. The highly significant p-value (less than 0.001) indicates the model is statistically significant, meaning there's a relationship between runs scored and salary that's unlikely due to chance.

This code builds and summarizes a linear regression model that explores the relationship between player runs scored and their salary using the training data. The results suggest a statistically significant positive relationship, where players with higher runs scored tend to have higher predicted salaries according to the model.

Evaluating the liner regression model for runs

Code used and Results:

```
> # Evaluate the model for runs
> y_pred_runs <- predict(model_runs, newdata = data.frame(runs_scored = X_test_runs$runs_scored))
> r2_runs <- cor(y_test_runs, y_pred_runs)^2
> print(paste("R-squared for runs: ", r2_runs))
[1] "R-squared for runs:  0.190229134838644"
```

Interpretation: The model's performance was evaluated using unseen data (testing set). The R-squared value of 0.19 indicates that the model explains about 19% of the variation in actual salaries in the testing data based on the number of runs scored. This is slightly lower than the R-squared obtained from the training data (24-25%), suggesting the model might perform a bit weaker on new data. Overall, this moderate ability to predict salaries based on runs scored

highlights the potential limitations of this model and indicates that other factors likely play a significant role in determining player salaries.

Creating the regression equation for the runs model

Code used and Results:

```
> # Extract the coefficients from the model_runs
> coefficients <- coef(model_runs)
> # Construct the equation
> equation <- paste0("y = ", round(coefficients[1], 2))
> for (i in 2:length(coefficients)) {
+   equation <- paste0(equation, " + ", round(coefficients[i], 6), "*x", i-1)
+ }
> # Print the equation
> print(equation)
[1] "y = 332.83 + 1.368985*x1"
```

$$y = 332.83 + 1.368985*x1$$

The above equation explains the relationship between the independent variable and t he dependent variable in the linear regression model for runs.

Interpretation: The key takeaway from this step is the linear regression equation: y = 332.83 + 1.368985*x1. This equation helps us understand how the model predicts salary (y) based on runs scored (x1). While the intercept (332.83) represents an unlikely scenario (zero runs scored), the coefficient for runs scored (1.368985) is more meaningful. It suggests that for every additional run a player scores, their predicted salary increases by 1.368985 units (depending on how salary is measured in the data). This equation provides a concise way to interpret the model's findings regarding the relationship between player performance (runs scored) and their salary.

The same steps from "***Creating linear regression model for runs"*** is used to create a linear regression model for predicting the salary of players (Bowler) based on the number of wickets taken by that bowler.

D. Regression – Predictive Analytics of IPL data using Python

Grouping and aggregating the performance metrics and creating the merged data table

Code used:

```
grouped_data = df_ipl.groupby(['Season', 'Innings No', 'Striker','Bowler']).agg({'runs_scored': sum, 'wicket_confirmation':sum}).reset_index()

total_runs_each_year = grouped_data.groupby(['Season', 'Striker'])['runs_scored'].sum().reset_index()
```

total_wicket_each_year = grouped_data.groupby(['Season', 'Bowler'])['wicket_confirmation'].sum().reset_index()

Output:

	Season	Striker	runs_scored
0	2012	A Ashish Reddy	35
1	2012	A Chandila	0
2	2012	A Mishra	10
3	2012	A Nehra	9
4	2012	AA Chavan	7
...	...	...	...
2812	2021	V Kohli	71
2813	2021	V Shankar	42
2814	2021	Virat Singh	11
2815	2021	WP Saha	8
2816	2021	Washington Sundar	18

2817 rows × 3 columns

Similarly, the data sub-set for bowlers and their number of wickets is also made.

Code used:

```
from fuzzywuzzy import process

# Convert to DataFrame
df_salary = salary.copy()
df_runs = total_runs_each_year.copy()

# Function to match names
def match_names(name, names_list):
    match, score = process.extractOne(name, names_list)
    return match if score >= 80 else None  # Use a threshold score of 80

# Create a new column in df_salary with matched names from df_runs
df_salary['Matched_Player'] = df_salary['Player'].apply(lambda x: match_names(x, df_runs['Striker'].tolist()))

# Merge the DataFrames on the matched names
df_merged = pd.merge(df_salary, df_runs, left_on='Matched_Player', right_on='Striker')
```

```
df_original = df_merged.copy()
```

```
#susbsets data for last three years
df_merged = df_merged.loc[df_merged['Season'].isin(['2021','2022','2023'])]
```

Output:

	Player	Salary	Rs	international	iconic	Matched_Player	Season	Striker	runs_scored
33	David Warner	6.25 crore	625	1	NaN	DA Warner	2021	DA Warner	93
76	Lalit Yadav	65 lakh	65	0	NaN	Lalit Yadav	2021	Lalit Yadav	32
98	Prithvi Shaw	7.5 crore	750	0	NaN	PP Shaw	2021	PP Shaw	106
107	Rishabh Pant	16 crore	1600	0	NaN	RR Pant	2021	RR Pant	81
129	Ajinkya Rahane	50 lakh	50	0	NaN	AM Rahane	2021	AM Rahane	8

Interpretation: This code is preparing data for further analysis, likely aiming to explore the relationship between player salary and their performance (runs scored) while accounting for potential inconsistencies in player names across datasets.

Creating linear regression model for runs scored

Code used and Results:

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
```

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_absolute_percentage_error
X = df_merged[['runs_scored']] # Independent variable(s)
y = df_merged['Rs'] # Dependent variable
# Split the data into training and test sets (80% for training, 20% for testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Create a LinearRegression model
model = LinearRegression()
# Fit the model on the training data
model.fit(X_train, y_train)
```

```
▾ LinearRegression
LinearRegression()
```

Interpretation: This code focuses on building a linear regression model to understand how players' runs are scored about their salary. It starts by preparing the data, importing necessary libraries and splitting it into training and testing sets. The training data is then used to fit a linear regression model that learns the relationship between runs scored and salary. While the code snippet doesn't show explicit evaluation, the analysis likely involves using the testing set to assess the model's generalizability and potentially calculating metrics like R-squared to measure how well the model explains salary variations based on runs scored.

Code used:

```
import pandas as pd
from sklearn.model_selection import train_test_split
import statsmodels.api as sm

# Assuming df_merged is already defined and contains the necessary columns
X = df_merged[['runs_scored']] # Independent variable(s)
y = df_merged['Rs'] # Dependent variable

# Split the data into training and test sets (80% for training, 20% for testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Add a constant to the model (intercept)
X_train_sm = sm.add_constant(X_train)

# Create a statsmodels OLS regression model
model = sm.OLS(y_train, X_train_sm).fit()

# Get the summary of the model
summary = model.summary()
print(summary)
```

Output:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                     Rs   R-squared:                       0.001
Model:                            OLS   Adj. R-squared:                 -0.028
Method:                 Least Squares   F-statistic:                   0.04842
Date:                Sun, 23 Jun 2024   Prob (F-statistic):              0.827
Time:                        23:31:52   Log-Likelihood:                -273.85
No. Observations:                  36   AIC:                             551.7
Df Residuals:                      34   BIC:                             554.9
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        729.5961    113.543      6.426      0.000     498.849     960.343
runs_scored    0.3719      1.690      0.220      0.827      -3.063       3.807
==============================================================================
Omnibus:                        2.654   Durbin-Watson:                   1.809
Prob(Omnibus):                  0.265   Jarque-Bera (JB):                2.393
Skew:                           0.606   Prob(JB):                        0.302
Kurtosis:                       2.647   Cond. No.                         91.4
==============================================================================
```

Interpretation: This analysis examined how well player runs scored and can predict their salary. The results suggest a weak connection. The R-squared value (0.001) is meager, indicating the model explains almost none of the salary variations. Further tests (F-statistic and coefficient p-values) also confirm this weak link. According to this analysis, runs scored by themselves don't statistically influence salary. Other factors likely play a much more significant role in determining player salaries.

Creating the regression equation

Code used and Results:

```
# Construct the equation
equation = "y = " + str(round(coefficients[0], 2))
for i in range(1, len(coefficients)):
  equation += " + " + str(round(coefficients[i], 6)) + "*x" + str(i)

# Print the equation
print(equation)
```

```
y = 346.6 + 148.796526*x1
```

$$y = 332.83 + 1.368985*x1$$

The above equation explains the relationship between the independent variable and t he dependent variable in the linear regression model for runs.

Interpretation: The key takeaway from this step is the linear regression equation: y = 332.83 + 1.368985*x1. This equation helps us understand how the model predicts salary (y) based on runs scored (x1). While the intercept (332.83) represents an unlikely scenario (zero runs scored), the coefficient for runs scored (1.368985) is more meaningful. It suggests that for every additional run a player scores, their predicted salary increases by 1.368985 units (depending on how salary is measured in the data). This equation provides a concise way to interpret the model's findings regarding the relationship between player performance (runs scored) and their salary.

The same steps from "***Creating linear regression model for runs"*** is used to create a linear regression model for predicting the salary of players (Bowler) based on the number of wickets taken by that bowler.