VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Data Mining

IT160IU

FINAL REPORT

MSc: Nguyen Quang Phu

DATA MINING

FOR WORLD HAPPINESS PREDICTION

BY GROUP DSA– MEMBER LIST

NGUYEN DU NHAN	ITDSIU22140	Leader
NGUYEN THE HAO	ITDSIU22139	Member
NGUYEN VAN HUY	ITITIU20215	Member

Table of Contents

I. Introduction

This project aims to develop a data mining framework that applies systematic analysis and machine learning to understand global well-being trends using the *World Happiness Report 2024* dataset. The objective is to build a classification and prediction model capable of identifying happiness levels based on socio-economic and emotional indicators such as *Log GDP per capital*, *Social Support*, *Healthy Life Expectancy*, *Freedom to make life choices*, *Generosity*, *Perceptions of corruption*, and emotional affect scores. The dataset spans multiple years and provides valuable insight into how different factors shape overall life satisfaction across countries. The project employs essential data mining concepts, such as data exploration, preprocessing, pattern discovery, and supervised learning, combined with machine learning algorithms such as *Random Forests* and *KNN*. These methods are significant because they reveal relationships among variables, support predictive analysis, and help transform raw data into meaningful knowledge that aligns with the goals of this assignment.

II. Data Pre-Processing

The goal of this stage is to clean and prepare the raw *World Happiness Report 2024* dataset for accurate analysis and modeling. This involves identifying and handling missing values, removing duplicates, addressing outliers, and applying necessary transformations such as encoding and scaling to ensure the data is consistent, reliable, and ready for machine learning tasks.

2.1 Raw Data Overview

The raw *World Happiness Report 2024* dataset contains:

```java
// ============================
// 1. Basic dataset information
// ============================
public static void printDatasetInfo(Instances data) {  1 usage   DuNhan1930
    System.out.println("=== DATASET INFO ===");
    System.out.println("Relation name : " + data.relationName());
    System.out.println("Num instances : " + data.numInstances());
    System.out.println("Num attributes: " + data.numAttributes());
    System.out.println("Class index   : " + data.classIndex()
            + " (" + (data.classIndex() >= 0 ? data.classAttribute().name() : "no class set") + ")");
    System.out.println();
}
```

Figure 2.1.1. Print Dataset Info

```
=== DATASET INFO ===
Relation name : World Happiness Report 2024
Num instances : 2363
Num attributes: 11
```

Figure 2.1.2. Result of Print Dataset Info

- 2,363 instances
- 11 attributes

A mix of economic, social, and emotional indicators collected across multiple years and countries. Key features include *Life Ladder*, *Log GDP per capital*, *Social support*, *Healthy life expectancy*, *Freedom to make life choices*, *Generosity*, *Perceptions of corruption*, *Positive affect*, and *Negative affect*.

A review of missing data shows:

```java
// ========================================
// 2. Missing values per attribute (null count)
// ========================================
public static void printMissingValues(Instances data) {  1 usage   DuNhan1930
    System.out.println("=== MISSING VALUES PER ATTRIBUTE ===");
    int n = data.numInstances();
    for (int a = 0; a < data.numAttributes(); a++) {
        Attribute att = data.attribute(a);
        int missing = 0;
        for (int i = 0; i < n; i++) {
            Instance inst = data.instance(i);
            if (inst.isMissing(a)) missing++;
        }
        double pct = (n == 0) ? 0.0 : (100.0 * missing / n);
        System.out.printf("Attribute %-35s : missing=%4d (%.2f%%)%n",
                att.name(), missing, pct);
    }
    System.out.println();
}
```

Figure 2.1.3. Print Missing Value

```
=== MISSING VALUES PER ATTRIBUTE ===
Attribute Country name                          : missing=   0 (0.00%)
Attribute year                                  : missing=   0 (0.00%)
Attribute Life Ladder                           : missing=   0 (0.00%)
Attribute Log GDP per capita                    : missing=  28 (1.18%)
Attribute Social support                        : missing=  13 (0.55%)
Attribute Healthy life expectancy at birth      : missing=  63 (2.67%)
Attribute Freedom to make life choices          : missing=  36 (1.52%)
Attribute Generosity                            : missing=  81 (3.43%)
Attribute Perceptions of corruption             : missing= 125 (5.29%)
Attribute Positive affect                       : missing=  24 (1.02%)
Attribute Negative affect                       : missing=  16 (0.68%)
```

Figure 2.1.4. Result of Print Missing Value

- No missing values in *Country name*, *year*, or *Life Ladder*.

Numerical features contain small-to-moderate missing rates:

- Lowest: *Social support* (0.55%)
- Highest: *Perceptions of corruption* (5.29%)

The overall missing percentage is low and manageable for imputation.

The dataset displays realistic ranges for all indicators:

```java
// ========================================
// 3. Numeric summaries: min, max, mean, median, std
// ========================================
public static void printNumericSummaries(Instances data, List<String> numericCols) {  1 usage   DuNhan1930
    System.out.println("=== NUMERIC SUMMARIES (min, max, mean, median, std) ===");
    for (String col : numericCols) {
        Attribute att = data.attribute(col);
        if (att == null) {
            System.out.println("Warning: numeric col not found: " + col);
            continue;
        }
        if (!att.isNumeric()) {
            System.out.println("Warning: col is not numeric: " + col);
            continue;
        }

        List<Double> vals = collectNonMissingValues(data, att.index());
        if (vals.isEmpty()) {
            System.out.println("No data for: " + col);
            continue;
        }

        Collections.sort(vals);
        double min = vals.get(0);
        double max = vals.get(vals.size() - 1);
        double mean = mean(vals);
        double median = median(vals);
        double std = stdDev(vals, mean);

        System.out.printf("Attribute %-35s : min=%10.4f | max=%10.4f | mean=%10.4f | median=%10.4f | std=%10.4f%n",
                col, min, max, mean, median, std);
    }
    System.out.println();
}
```

Figure 2.1.5. Code Print Numeric Summaries

- *Life Ladder* ranges from 1.28 to 8.02
- *Log GDP per capital* ranges from 5.53 to 11.68
- *Socio-emotional* indicators (*Social support, Positive affect, etc.*) show consistent global variation.

Most variables are near-symmetric, though some show skewness:

```java
// ========================================
// 6. Skewness per numeric attribute
// ========================================
public static void printSkewness(Instances data, List<String> numericCols) {  1 usage   DuNhan1930 *
    System.out.println("=== SKEWNESS PER ATTRIBUTE ===");
    System.out.println("Note: |skew| < 0.5 ≈ symmetric, 0.5-1 ≈ moderate, > 1 = highly skewed");
    System.out.println();

    for (String col : numericCols) {
        Attribute att = data.attribute(col);
        if (att == null) {
            System.out.println("Warning: numeric col not found: " + col);
            continue;
        }
        if (!att.isNumeric()) {
            System.out.println("Warning: col is not numeric: " + col);
            continue;
        }

        List<Double> vals = collectNonMissingValues(data, att.index());
        if (vals.isEmpty()) {
            System.out.println("No data for: " + col);
            continue;
        }

        double sk = skewness(vals);
        double absSk = Math.abs(sk);
        String level;
        if (Double.isNaN(sk)) {
            level = "NA (not enough data)";
        } else if (absSk < 0.5) {
            level = "≈ symmetric";
        } else if (absSk < 1.0) {
            level = "moderately skewed";
        } else {
            level = "highly skewed";
        }

        System.out.printf("Attribute %-35s : skewness=%8.4f → %s%n",
                col, sk, level);
    }
    System.out.println();
}
```

Figure 2.1.6. Print Skewness per Attribute

```
=== SKEWNESS PER ATTRIBUTE ===
Note: |skew| < 0.5 ≈ symmetric, 0.5-1 ≈ moderate, > 1 = highly skewed

Attribute year                              : skewness= -0.0643 → ≈ symmetric
Attribute Life Ladder                       : skewness= -0.0538 → ≈ symmetric
Attribute Log GDP per capita                : skewness= -0.3365 → ≈ symmetric
Attribute Social support                    : skewness= -1.1087 → highly skewed
Attribute Healthy life expectancy at birth  : skewness= -1.1291 → highly skewed
Attribute Freedom to make life choices      : skewness= -0.6994 → moderately skewed
Attribute Generosity                        : skewness=  0.7691 → moderately skewed
Attribute Perceptions of corruption         : skewness= -1.4847 → highly skewed
Attribute Positive affect                   : skewness= -0.4588 → ≈ symmetric
Attribute Negative affect                   : skewness=  0.6978 → moderately skewed
```

Figure 2.1.7. Result of Skewness per Attribute

- Highly skewed: *Social support*, *Healthy life expectancy*, *Perceptions of corruption*
- Moderately skewed: *Freedom*, *Generosity*, *Negative affect*

Outlier detection reveals that several attributes contain outliers:

```java
// ========================================
// 5. Outlier statistics via Tukey's fences
// ========================================
public static void printOutlierStats(Instances data, List<String> numericCols) {  1 usage   DuNhan1930
    System.out.println("=== OUTLIER STATS (Tukey fences) ===");

    for (String col : numericCols) {
        Attribute att = data.attribute(col);
        if (att == null) {
            System.out.println("Warning: numeric col not found: " + col);
            continue;
        }
        if (!att.isNumeric()) {
            System.out.println("Warning: col is not numeric: " + col);
            continue;
        }

        List<Double> vals = collectNonMissingValues(data, att.index());
        if (vals.isEmpty()) {
            System.out.println("No data for: " + col);
            continue;
        }
        Collections.sort(vals);
        double q1 = percentile(vals,  p: 25.0);
        double q3 = percentile(vals,  p: 75.0);
        double iqr = q3 - q1;
        double lw = q1 - 1.5 * iqr;
        double uw = q3 + 1.5 * iqr;

        int lowCount = 0;
        int highCount = 0;

        for (double v : vals) {
            if (v < lw) lowCount++;
            else if (v > uw) highCount++;
        }

        System.out.printf("Attribute %-35s : Q1=%8.4f | Q3=%8.4f | IQR=%8.4f | lw=%8.4f | uw=%8.4f | lowOut=%4d | highOut=%4d%n",
                col, q1, q3, iqr, lw, uw, lowCount, highCount);
    }
    System.out.println();
}
```

Figure 2.1.8. Print Outlier Statistics

```
=== OUTLIER STATS (Tukey fences) ===
Attribute year                              : Q1=2011.0000 | Q3=2019.0000 | IQR=  8.0000 | lw=1999.0000 | uw=2031.0000 | lowOut=   0 | highOut=   0
Attribute Life Ladder                       : Q1=  4.6467 | Q3=  6.3236 | IQR=  1.6768 | lw=  2.1315 | uw=  8.8389 | lowOut=   2 | highOut=   0
Attribute Log GDP per capita                : Q1=  8.5062 | Q3= 10.3930 | IQR=  1.8868 | lw=  5.6760 | uw= 13.2232 | lowOut=   1 | highOut=   0
Attribute Social support                    : Q1=  0.7438 | Q3=  0.9038 | IQR=  0.1600 | lw=  0.5039 | uw=  1.1437 | lowOut=  49 | highOut=   0
Attribute Healthy life expectancy at birth  : Q1= 59.1950 | Q3= 68.5525 | IQR=  9.3575 | lw= 45.1587 | uw= 82.5888 | lowOut=  20 | highOut=   0
Attribute Freedom to make life choices      : Q1=  0.6607 | Q3=  0.8617 | IQR=  0.2010 | lw=  0.3592 | uw=  1.1633 | lowOut=  16 | highOut=   0
Attribute Generosity                        : Q1= -0.1119 | Q3=  0.0936 | IQR=  0.2055 | lw= -0.4202 | uw=  0.4018 | lowOut=   0 | highOut=  39
Attribute Perceptions of corruption         : Q1=  0.6868 | Q3=  0.8676 | IQR=  0.1808 | lw=  0.4156 | uw=  1.1387 | lowOut= 195 | highOut=   0
Attribute Positive affect                   : Q1=  0.5720 | Q3=  0.7373 | IQR=  0.1653 | lw=  0.3240 | uw=  0.9852 | lowOut=   8 | highOut=   0
Attribute Negative affect                   : Q1=  0.2086 | Q3=  0.3262 | IQR=  0.1177 | lw=  0.0321 | uw=  0.5027 | lowOut=   0 | highOut=  29
```

Figure 2.1.9. Result of Outlier Statistics

- *Life Ladder*: 2 low outliers
- *Social support*: 49 low outliers

- *Perceptions of corruption*: 195 low outliers

These outliers reflect natural differences between countries rather than data errors.

The correlation matrix reveals clear relationships:

```java
// ========================================
// 4. Correlation matrix (Pearson) for selected numeric columns
// ========================================
public static void printCorrelationMatrix(Instances data, List<String> numericCols) {  1 usage   DuNhan1930 *
    System.out.println("=== CORRELATION MATRIX (Pearson) ===");
    int m = numericCols.size();
    int[] attIdx = new int[m];

    for (int i = 0; i < m; i++) {
        String col = numericCols.get(i);
        Attribute att = data.attribute(col);
        if (att == null || !att.isNumeric()) {
            System.out.println("Warning: cannot correlate, invalid numeric col: " + col);
            return;
        }
        attIdx[i] = att.index();
    }
    double[][] corr = new double[m][m];

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            if (i == j) {
                corr[i][j] = 1.0;
            } else if (j < i) {
                corr[i][j] = corr[j][i]; // symmetric
            } else {
                corr[i][j] = pearsonCorr(data, attIdx[i], attIdx[j]);
            }
        }
    }
    // print header
    System.out.print(String.format("%-28s", ""));
    for (int j = 0; j < m; j++) {
        System.out.print(String.format("%-12s", numericCols.get(j)));
    }
    System.out.println();
    // print rows
    for (int i = 0; i < m; i++) {
        System.out.print(String.format("%-28s", numericCols.get(i)));
        for (int j = 0; j < m; j++) {
            System.out.print(String.format("%-12.3f", corr[i][j]));
        }
        System.out.println();
```

Figure 2.1.10. Print Correlation Matrix

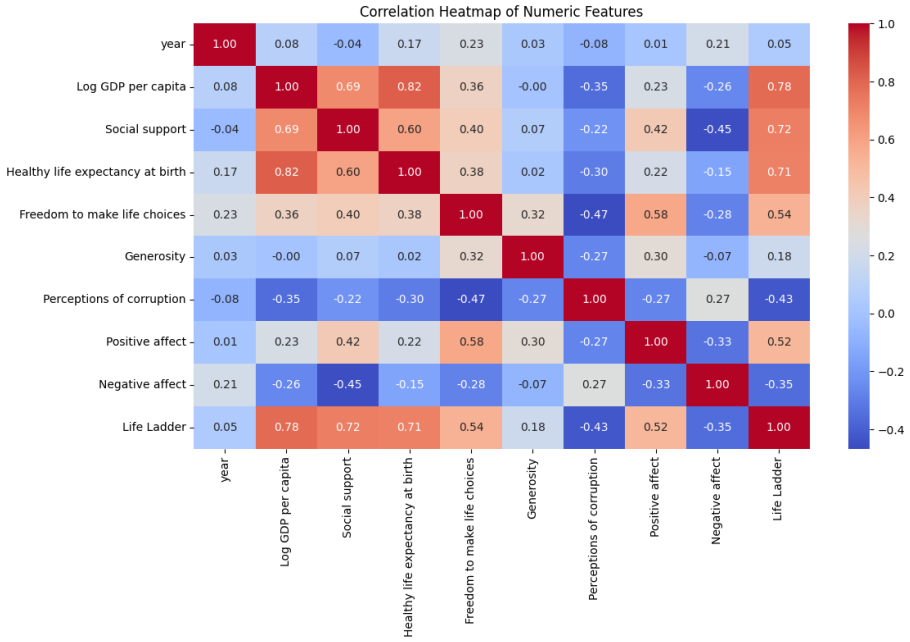

Figure 2.1.11. Correlation Matrix

Life Ladder strongly correlates with:

- *Log GDP per capital* (0.784)
- *Social support* (0.723)
- *Healthy life expectancy at birth* (0.715)

Significant high correlation among attributes:

- *Log GDP per capital* and *Healthy life expectancy at birth* (0.819)

These patterns suggest that economic and social factors significantly influence happiness scores.

2.2 Data Cleaning Process

2.2.1 Handling missing values

Several numerical attributes contained small-to-moderate missing rates (0.55%5.29%). To ensure consistency without distorting the distribution, missing values were replaced using the *median*.

```java
public class MissingValueHandler {  1 usage   DuNhan1930

    /**
     * Impute missing numeric values in the given columns using median.
     */
    public static void imputeMedian(Instances data, List<String> columns) {  1 usage   DuNhan1930
        for (String col : columns) {
            Attribute att = data.attribute(col);
            if (att == null) {
                System.out.println("Warning: column not found for median: " + col);
                continue;
            }
            if (!att.isNumeric()) {
                System.out.println("Warning: non-numeric median column: " + col);
                continue;
            }

            int idx = att.index();
            List<Double> vals = new ArrayList<>();
            for (int i = 0; i < data.numInstances(); i++) {
                Instance inst = data.instance(i);
                if (!inst.isMissing(idx)) {
                    vals.add(inst.value(idx));
                }
            }
            if (vals.isEmpty()) {
                System.out.println("Warning: all values missing for " + col);
                continue;
            }

            Collections.sort(vals);
            double median = medianOfList(vals);

            for (int i = 0; i < data.numInstances(); i++) {
                Instance inst = data.instance(i);
                if (inst.isMissing(idx)) {
                    inst.setValue(idx, median);
                }
            }

            System.out.println("Median-imputed: " + col + " (median=" + median + ")");
        }
    }
```

Figure 2.2.1. Handle Missing Value

The *median* is preferred over the mean because:

- It is robust against skewed distributions, which several attributes showed (*e.g., Social support, Perceptions of corruption*).
- It is not affected by outliers, ensuring more stable central values.

This approach preserves the natural variability of the dataset while preventing extreme values from pulling the imputed values upward or downward.

2.2.2 Removing duplicates

There are not duplicates.

2.2.3 Addressing outliers

Outlier detection using *Tukey fences* revealed multiple low and high outliers across attributes. Since these outliers likely represent genuine country-level differences rather than data errors, they were kept. However, to reduce the impact of extreme values during modeling, we applied *Robust Scaler*:

$$X_{scaled} = \frac{X - Median(X)}{Q_3 - Q_1}$$

Where:

- X is the original value
- $Median(X)$ is the Q_2or the median quartile (50th percentile)
- Q_1 is the first quartile (25th percentile)
- Q_3 is the third quartile (75th percentile)

It is well-suited because:

- It uses the *median* and *IQR* instead of *mean* and *standard deviation*.
- It is resistant to the influence of outliers, ensuring the model does not become biased by extreme values.

This scaling method helps stabilize the distribution of features and supports better training of machine learning models.

```java
public static void robustScale(Instances data, List<String> numericCols) {  1 usage  DuNhan1930 *
    for (String col : numericCols) {
        Attribute att = data.attribute(col);
        if (att == null) {
            System.out.println("Warning: column not found for robust scaling: " + col);
            continue;
        }
        if (!att.isNumeric()) {
            System.out.println("Warning: non-numeric robust scaling column: " + col);
            continue;
        }
        int idx = att.index();
        // Collect non-missing values
        List<Double> vals = new ArrayList<>();
        for (int i = 0; i < data.numInstances(); i++) {
            Instance inst = data.instance(i);
            if (!inst.isMissing(idx)) {
                vals.add(inst.value(idx));
            }
        }
        if (vals.isEmpty()) {
            System.out.println("Warning: no numeric data for " + col);
            continue;
        }
        // Sort for median + percentiles
        Collections.sort(vals);
        double median = medianOfList(vals);
        double q1 = percentile(vals,  p: 25.0);
        double q3 = percentile(vals,  p: 75.0);
        double iqr = q3 - q1;
        if (Double.isNaN(median) || Double.isNaN(iqr) || iqr == 0.0) {
            System.out.println("Warning: invalid IQR (" + iqr + ") for " + col + ", skip robust scaling.");
            continue;
        }
        // Apply (x - median) / IQR to all non-missing values
        for (int i = 0; i < data.numInstances(); i++) {
            Instance inst = data.instance(i);
            if (inst.isMissing(idx)) continue;
            double x = inst.value(idx);
            double scaled = (x - median) / iqr;
            inst.setValue(idx, scaled);
        }
        System.out.println("Robust scaled " + col + " (median=" + median + ", IQR=" + iqr + ")");
    }
```

Figure 2.2.2. Robust Scaler

2.3 Data Transformation

2.3.1 Creating the *Life Ladder Category*

To enable classification modeling, the continuous variable Life Ladder was transformed into a categorical label called *Life Ladder Category*. This label was created using defined bins:

- *Low*: 0-5
- *Medium*: 5-7
- *High*: 7-10

This transformation converts a numerical indicator of subjective well-being into meaningful classes that machine learning algorithms can classify. It also supports easier interpretation of results by grouping countries into distinct happiness levels rather than using raw continuous values.

```java
private static void ensureLifeLadderCategory(Instances data) {  1 usage  DuNhan1930
    Attribute catAttr = data.attribute(LIFE_LADDER_CAT_COL);
    if (catAttr != null) {
        System.out.println("Life Ladder Category already exists.");
        return;
    }

    Attribute lifeLadderAttr = data.attribute(LIFE_LADDER_NUM_COL);
    if (lifeLadderAttr == null || !lifeLadderAttr.isNumeric()) {
        throw new IllegalArgumentException("Numeric attribute '" + LIFE_LADDER_NUM_COL + "' not found.");
    }

    // Create nominal attribute (Low, Medium, High, Very High)
    java.util.ArrayList<String> levels = new java.util.ArrayList<>();
    levels.add("Low");
    levels.add("Medium");
    levels.add("High");

    Attribute newCatAttr = new Attribute(LIFE_LADDER_CAT_COL, levels);
    int newIndex = data.numAttributes();
    data.insertAttributeAt(newCatAttr, newIndex);

    int lifeIdx = lifeLadderAttr.index();
    int catIdx  = data.attribute(LIFE_LADDER_CAT_COL).index();

    for (int i = 0; i < data.numInstances(); i++) {
        Instance inst = data.instance(i);
        if (inst.isMissing(lifeIdx)) {
            inst.setMissing(catIdx);
            continue;
        }
        double v = inst.value(lifeIdx);
        String label;
        if (v <= 5.0) {
            label = "Low";
        } else if (v <= 7.0) {
            label = "Medium";
        } else {
            label = "High";
        }
        inst.setValue(catIdx, label);
    }

    System.out.println("Created Life Ladder Category attribute.");
}
```

Figure 2.3.1. Create Life Ladder Category

2.3.2 Encoding the *Country Name* (*NominalToBinary*)

The attribute Country name is categorical and contains many unique country labels. To make this attribute usable for classification algorithms, it was transformed using *one-hot encoding* through *Weka's NominalToBinary* filter.

This transformation is suitable because:

- Most machine learning models require numeric input, and cannot directly handle string categories.
- *One-hot encoding* avoids misleading ordinal relationships that would occur if countries were encoded as integers.
- Each country becomes a separate binary feature, allowing the model to learn country-level patterns without bias or distortion.

```java
public static Instances oneHotEncodeNominal(Instances data, String attrName) throws Exception {
    Attribute att = data.attribute(attrName);
    if (att == null) {
        System.out.println("Warning: cannot one-hot, attribute not found: " + attrName);
        return data;
    }
    if (!att.isNominal()) {
        System.out.println("Warning: attribute not nominal for one-hot: " + attrName);
        return data;
    }

    int oneBasedIndex = att.index() + 1; // Weka filters use 1-based index strings
    String idxStr = String.valueOf(oneBasedIndex);

    NominalToBinary ntb = new NominalToBinary();
    ntb.setAttributeIndices(idxStr);     // only this attribute
    ntb.setBinaryAttributesNominal(false); // numeric 0/1

    ntb.setInputFormat(data);
    Instances newData = Filter.useFilter(data, ntb);

    System.out.println("One-hot encoded attribute: " + attrName);
    return newData;
}
```

Figure 2.3.2. One Hot Encode Category Attributes

```java
// 6. One-hot encode Country name (NominalToBinary)
// =====================
data = EncodingHandler.oneHotEncodeNominal(data, COUNTRY_COL);
```

Figure 2.3.3. Encode Country Name Attribute

2.3.3 Removing redundant attributes

The feature *Healthy life expectancy at birth* showed:

- Very strong correlation with *Log GDP per capital* (0.819)
- Very strong correlation with *Life Ladder* (0.715)

Because this attribute provides information already captured by *GDP* and other features, it introduces multicollinearity, which can reduce model interpretability and inflate feature importance.

Additionally, when applying dimensionality reduction concepts such as *PCA*, highly correlated variables contribute redundant variance and lower the efficiency of the feature space.

Therefore, removing *Healthy life expectancy at birth* helps:

- Reduce redundancy
- Improve model stability
- Enhance the effectiveness of later feature transformations

This decision keeps the dataset compact while maintaining the essential predictive signals.

```
public static void dropColumns(Instances data, List<String> colNames) {  1 usage  DuNhan19
    // Delete from last to first to keep indexes valid
    for (String name : colNames) {
        Attribute att = data.attribute(name);
        if (att == null) {
            System.out.println("Warning: cannot drop, attribute not found: " + name);
            continue;
        }
        int idx = att.index();
        data.deleteAttributeAt(idx);
        System.out.println("Dropped attribute: " + name);
    }
}
```

Figure 2.3.4. Drop Columns

```
// 5. Drop unwanted columns
// ======================
DropColumn.dropColumns(data, Arrays.asList(
        "Healthy life expectancy at birth",
        LIFE_LADDER_NUM_COL
));

System.out.println("After drop: " + data.numAttributes() + " attributes.");
```

Figure 2.3.5. Drop Columns

Output

Converting the processed dataset into *ARFF format*

```
// 8. Save to ARFF
// ======================
ArffSaver saver = new ArffSaver();
saver.setInstances(data);
saver.setFile(new File(OUTPUT_ARFF));
saver.writeBatch();

System.out.println("Saved preprocessed ARFF to: " + OUTPUT_ARFF);
System.out.println("Done.");
```

Figure 2.3.6. Convert to ARFF Format

The final cleaned dataset:

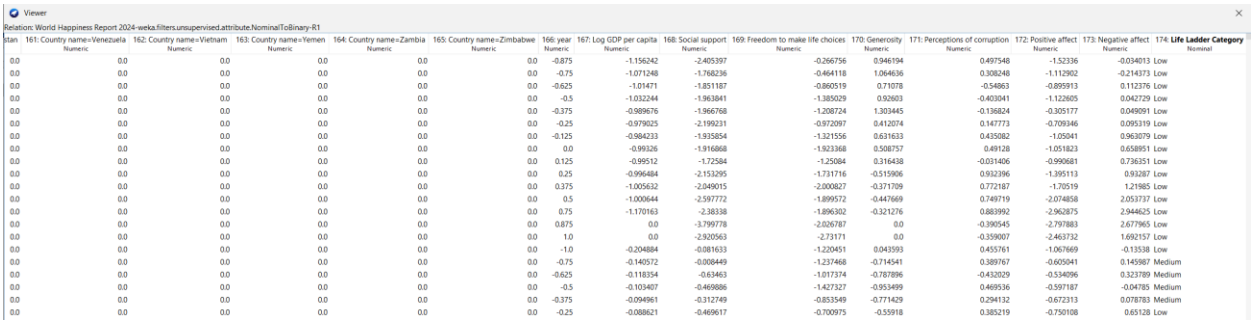

Figure 2.3.7 Final Cleaned Dataset

- 2,363 instances

- 174 attributes

III. Classification/Prediction Algorithm

The aim of this stage is to build and evaluate a classification model using the Weka library. This involves selecting an appropriate machine learning algorithm, and implementing the model to predict happiness categories based on the transformed features.

3.1 Model Selection

Random Forest Algorithm

Random Forest is an ensemble learning algorithm that builds multiple decision trees and combines their predictions to produce a more stable and accurate result. Each tree is trained on a randomly selected subset of the data and a random subset of features. During prediction, the algorithm aggregates the outputs of all trees - often by majority voting for classification tasks. This process reduces variance, improves generalization, and minimizes the risk of overfitting, which is a common issue in single decision trees.

Random Forest Algorithm - Pseudocode Description:

```
RandomForest(Dataset D, number_of_trees T):
  For i = 1 to T:
    1. Draw a bootstrap sample Di from the original dataset D.
    2. Grow a decision tree Ti using Di:
      a. At each split, randomly select a subset of features F.
      b. Choose the best feature from F based on impurity measure
        (e.g., Gini index or entropy).
      c. Split the node and continue recursively until stopping criteria.
  To classify a new instance x:
    1. For each tree Ti in the forest:
        predict class Ci(x)
    2. Return the class most frequently predicted
```

(majority voting across all trees).

Random Forest was selected for this project because it is highly effective for tabular datasets like the *World Happiness Report 2024*. The algorithm handles both numerical and categorical features well, is robust to noise and outliers, and performs reliably even when the dataset contains skewed distributions or missing patterns that have been imputed. Its built-in feature randomness improves model stability, while ensemble aggregation typically yields higher accuracy compared to individual decision trees. Additionally, *Random Forest* provides useful insights such as feature importance, helping identify which socio-economic factors contribute most to happiness classification. These characteristics make it an appropriate and reliable choice for the prediction task in this assignment.

```
private static RandomForest createRandomForest(Instances data, int numTrees) {  2 usages  DuNhan1930
    RandomForest rf = new RandomForest();

    rf.setNumIterations(numTrees);

    // Number of features per split = sqrt(#attributes - class)
    int numAttributes = data.numAttributes() - 1; // exclude class attribute
    int k = (int) Math.round(Math.sqrt(numAttributes));
    if (k < 1) k = 1;
    rf.setNumFeatures(k);

    rf.setSeed(36);

    // Commented out to reduce noise during loops, uncomment if needed
    // System.out.println("Configured RandomForest: trees=" + rf.getNumIterations() + ", features per split=" + rf.getNumFeatures());
    return rf;
}
```

Figure 3.1.1. Random Forest method

***K-Nearest Neighbors* Algorithm**

To improve the baseline *Random Forest* model, the *K-Nearest Neighbors* (*KNN*) algorithm was explored as an alternative classification approach. *KNN* is a distance-based, instance-based learning algorithm that makes predictions by examining the *k* closest data points in the feature space and assigning the most common class among them. It does not build an explicit model; instead, it relies on the structure of the dataset and the similarity between instances.

```
private static IBk createKNN(int k) {  2 usages   DuNhan1930
    IBk knn = new IBk();
    knn.setKNN(k);
    // Commented out to reduce noise during loops, uncomment if needed
    // System.out.println("Configured KNN with k = " + knn.getKNN());
    return knn;
}
```

Figure 3.1.2. KNN method

3.2 Implementation Process

Challenges faced during implementation:

- Ensuring proper conversion from *CSV* to *ARFF*, especially after applying *one-hot encoding*, which increased the number of attributes significantly.
- Managing attribute types in *Weka*, as some filters and classifiers required numeric-only or nominal-only inputs.
- Tuning *Random Forest and K-Nearest Neighbors hyperparameters* (e.g., number of trees, feature subset size, number of K) to achieve stable and accurate performance.

3.3 Results

3.3.1 Random Forest

The *Random Forest classifier* was evaluated using *10-fold cross-validation* on the transformed dataset containing 2,363 instances and 174 attributes, with *Life Ladder Category* as the target variable (3 classes). A hyperparameter search was conducted by testing different numbers of trees while keeping the number of features per split fixed at 13. The best performance was achieved with 50 trees, giving the highest cross-validation accuracy of 0.8722.

```
=== 10-fold CV: Random Forest hyper parameter search ===
Configured RandomForest: trees=50, features per split=13
Random Forest with 50 trees → CV Accuracy = 0.8722
Configured RandomForest: trees=100, features per split=13
Random Forest with 100 trees → CV Accuracy = 0.8663
Configured RandomForest: trees=150, features per split=13
Random Forest with 150 trees → CV Accuracy = 0.8697
Configured RandomForest: trees=200, features per split=13
Random Forest with 200 trees → CV Accuracy = 0.8680
Configured RandomForest: trees=300, features per split=13
Random Forest with 300 trees → CV Accuracy = 0.8671
Configured RandomForest: trees=400, features per split=13
Random Forest with 400 trees → CV Accuracy = 0.8650
Configured RandomForest: trees=500, features per split=13
Random Forest with 500 trees → CV Accuracy = 0.8697

[Random Forest] Best numTrees = 50 with CV Accuracy = 0.8722
```

Figure 3.3.1. Result of Random Forest Tuning

Overall Performance:

- Best Model: *Random Forest* (50 trees)
- Cross-Validation Accuracy: 0.8722
- Weighted F1-score: 0.8719
- Runtime: 5m20s

```
=======================================
Model: Random Forest (best=50 trees) (10-fold CV)
---------------------------------------
Per-fold Accuracy Details:
Fold  1: 82.7004% (237 instances)
Fold  2: 85.6540% (237 instances)
Fold  3: 87.7637% (237 instances)
Fold  4: 89.4068% (236 instances)
Fold  5: 85.1695% (236 instances)
Fold  6: 82.6271% (236 instances)
Fold  7: 89.4068% (236 instances)
Fold  8: 87.2881% (236 instances)
Fold  9: 88.5593% (236 instances)
Fold 10: 87.2881% (236 instances)
---------------------------------------
Average Accuracy (CV): 0.8658
Weighted F1-score (CV): 0.8655
```

Figure 3.3.2. Result of Random Forest in each Fold

```
Model: Random Forest (best=50 trees) (10-fold CV)
Accuracy (CV): 0.8722
Weighted F1-score (CV): 0.8719

Per-class metrics (precision, recall, F1, support):
Class Low        | precision=0.8670 | recall=0.8690 | f1=0.8680 | support=840
Class Medium     | precision=0.8708 | recall=0.8929 | f1=0.8818 | support=1261
Class High       | precision=0.8991 | recall=0.7824 | f1=0.8367 | support=262

Confusion matrix (rows=true, cols=pred):
          Low       Medium    High
Low       730       110       0
Medium    112       1126      23
High      0         57        205
```

Figure 3.3.3. Evaluation Metric and Confusion Matrix of Random Forest

The model performs consistently well across all classes, with the *Medium* and *Low* categories achieving strong recall values. The *High* category has slightly lower recall due to its smaller support, but the precision remains high. Overall, the *Random Forest* model demonstrates strong predictive capability for classifying happiness levels based on socio-economic and emotional indicators.

3.3.2 K-Nearest Neighbors

For this project, multiple values of *k* were tested using *10-fold cross-validation*. Smaller values of k allow the classifier to be more sensitive to local patterns, while larger values smooth the decision boundary. The best performance was achieved at $k = 5$, which provided a good balance between stability and sensitivity to local variations.

```
=== 10-fold CV: KNN hyper parameter search ===
Configured KNN with k = 1
KNN with k=1 → CV Accuracy = 0.8764
Configured KNN with k = 3
KNN with k=3 → CV Accuracy = 0.8735
Configured KNN with k = 5
KNN with k=5 → CV Accuracy = 0.8764
Configured KNN with k = 7
KNN with k=7 → CV Accuracy = 0.8739
Configured KNN with k = 9
KNN with k=9 → CV Accuracy = 0.8570
Configured KNN with k = 11
KNN with k=11 → CV Accuracy = 0.8506
Configured KNN with k = 13
KNN with k=13 → CV Accuracy = 0.8477

[KNN] Best k = 5 with CV Accuracy = 0.8764
Configured RandomForest: trees=50, features per split=13
Configured KNN with k = 5
```

Figure 3.3.4. Result of KNN Tuning

Overall Performance:

- Best Model: *KNN* (k=5)
- Cross-Validation Accuracy: 0.8764
- Weighted F1-score: 0.8764
- Run time: 36s

```
========================================
Model: KNN (best k=5) (10-fold CV)
----------------------------------------
Per-fold Accuracy Details:
Fold  1: 84.8101% (237 instances)
Fold  2: 89.8734% (237 instances)
Fold  3: 88.1857% (237 instances)
Fold  4: 87.2881% (236 instances)
Fold  5: 87.2881% (236 instances)
Fold  6: 85.5932% (236 instances)
Fold  7: 89.8305% (236 instances)
Fold  8: 86.8644% (236 instances)
Fold  9: 87.7119% (236 instances)
Fold 10: 88.9831% (236 instances)
----------------------------------------
Average Accuracy (CV): 0.8760
Weighted F1-score (CV): 0.8760
```

Figure 3.3.5. Result of KNN in each Fold

```
Model: KNN (best k=5) (10-fold CV)
Accuracy (CV): 0.8764
Weighted F1-score (CV): 0.8764

Per-class metrics (precision, recall, F1, support):
Class Low        | precision=0.8730 | recall=0.8917 | f1=0.8822 | support=840
Class Medium     | precision=0.8898 | recall=0.8771 | f1=0.8834 | support=1261
Class High       | precision=0.8244 | recall=0.8244 | f1=0.8244 | support=262

Confusion matrix (rows=true, cols=pred):
          Low       Medium    High
Low       749       91        0
Medium    109       1106      46
High      0         46        216
```

Figure 3.3.6. Evaluation Metric and Confusion Matrix of KNN

KNN slightly outperformed the initial *Random Forest* model (*accuracy* 0.8764 vs. 0.8722). Its strong performance indicates that the dataset has well-separated feature clusters, and similarity-based decision boundaries work effectively for predicting happiness categories.

IV. Improvement of Results

The goal of this section is to improve the initial classification results by applying alternative machine learning algorithms to enhance model accuracy, stability, and overall predictive performance.

4.1 Methodology

Multi-layer Perceptron Algorithm

To address the high dimensionality of the dataset (173 input features) and improve classification accuracy, we implemented a custom *Multi-Layer Perceptron* (MLP) neural network. This deep learning approach was chosen to capture complex, non-linear relationships between the happiness indicators that simpler linear models might miss.

The improved model incorporates several advanced architectural and optimization techniques.

4.1.1 Optimized Network Architecture

We designed a feed-forward neural network with a "funnel" architecture to progressively compress features:

- Input Layer: 173 neurons, matching the preprocessed dataset attributes.
- Hidden Layer 1: 128 neurons with ReLU (Rectified Linear Unit) activation to mitigate the vanishing gradient problem.
- Hidden Layer 2: 64 neurons with ReLU activation.
- Output Layer: 3 neurons using Softmax activation to output a probability distribution across the three target classes (Low, Medium, High).

```
public Network(int inputSize) {  2 usages  DuNhan1930
    // --- Architecture for High-Dimensional Data (~173 inputs) ---
    // Input -> Hidden 1 (128 neurons) -> Hidden 2 (64 neurons) -> Output (3 classes)
    layers.add(new Layer(inputSize,  numNeurons: 128));
    layers.add(new Layer( numInputs: 128,  numNeurons: 64));
    layers.add(new Layer( numInputs: 64,  numNeurons: 3));
}
```

Figure 4.1.1. Network Config

4.1.2 Advanced Optimization (Adam)

Instead of standard *Stochastic Gradient Descent* (SGD), we implemented the Adam (*Adaptive Moment Estimation*) optimizer. Adam computes adaptive learning rates for each parameter by estimating the first moment (mean) and second moment (variance) of the gradients. This allows the model to converge faster and navigate the complex error surface more effectively.

4.1.3 Regularization & Generalization

To prevent overfitting, which is a critical risk given the 173 input features, we applied three specific regularization strategies:

- **Dropout**: We implemented Inverted Dropout with a rate of $p = 0.2$. During training, 20% of hidden neurons are randomly zeroed out, forcing the network to learn robust, redundant feature representations.
- **L2 Regularization (Weight Decay)**: We added an L2 penalty term (λ) to the weight updates, which penalizes large weights and encourages smoother decision boundaries.
- **He Initialization**: Weights were initialized using He Initialization ($\mathcal{N}\left(0, \sqrt{2/n_{in}}\right)$), which is mathematically optimal for ReLU networks to maintain variance in the forward pass.

4.1.4 Training Strategy (Early Stopping)

We implemented an Early Stopping mechanism with Weight Restoration. The training process utilizes a validation set (10% of training data) to monitor performance. If validation loss does not improve for 20 consecutive epochs (patience), training is halted, and the model restores the weights from the epoch with the lowest recorded loss. This ensures the final model represents the point of best generalization rather than the point of lowest training error.

```
    if (best_val_loss - val_loss > MIN_DELTA) {
        best_val_loss = val_loss;
        best_epoch = epoch;
        bestModel.copyWeightsFrom(model);
        wait = 0;

        // Just print log when have new record
        if (epoch % 5 == 0) {
            System.out.printf("   [+] Epoch %d: New Best Loss %.5f\n", epoch, val_loss);
        }
    } else {
        wait++;
    }

    // Early Stopping Condition
    if (epoch > MIN_EPOCHS_BEFORE_STOP && wait >= PATIENCE) {
        System.out.println("   [STOP] Early Stopping triggered at Epoch " + epoch);
        break;
    }
}
```

Figure 4.1.2. Early Stopping Algorithm

```
=== Fold 1 ===
   [+] Epoch 5: New Best Loss 0.39326
   [+] Epoch 15: New Best Loss 0.36821
   [STOP] Early Stopping triggered at Epoch 52
   [RESTORE] Loading weights from Best Epoch: 32
-> Fold 1 Test Accuracy: 89.87%

=== Fold 2 ===
   [+] Epoch 5: New Best Loss 0.37015
   [+] Epoch 10: New Best Loss 0.34326
   [+] Epoch 15: New Best Loss 0.34111
   [+] Epoch 25: New Best Loss 0.32890
   [+] Epoch 35: New Best Loss 0.31684
   [STOP] Early Stopping triggered at Epoch 82
   [RESTORE] Loading weights from Best Epoch: 62
-> Fold 2 Test Accuracy: 92.41%
```

Figure 4.1.3. Example Result of Early Stopping

4.2 Comparison of Results

To assess the efficacy of the proposed Deep Learning approach, we compared the custom *Multi-Layer Perceptron* (MLP) against two established baseline algorithms: Random Forest (an ensemble method) and K-Nearest Neighbors (KNN) (an instance-based method). All models were evaluated using the same 10-Fold Cross-Validation protocol to ensure fair comparison.

```
=== Fold 2 ===
   [+] Epoch 5: New Best Loss 0.37015
   [+] Epoch 10: New Best Loss 0.34326
   [+] Epoch 15: New Best Loss 0.34111
   [+] Epoch 25: New Best Loss 0.32890
   [+] Epoch 35: New Best Loss 0.31684
   [STOP] Early Stopping triggered at Epoch 82
   [RESTORE] Loading weights from Best Epoch: 62
-> Fold 2 Test Accuracy: 92.41%
```

Figure 4.2.1. Best Fold Result of MLP

```
=== Fold 3 ===
   [+] Epoch 5: New Best Loss 0.46939
   [+] Epoch 15: New Best Loss 0.44391
   [+] Epoch 20: New Best Loss 0.43513
   [STOP] Early Stopping triggered at Epoch 67
   [RESTORE] Loading weights from Best Epoch: 47
-> Fold 3 Test Accuracy: 84.81%
```

Figure 4.2.2. Worst Fold Result of MLP

```
Final Average Test Accuracy: 88.06%

=== Overall Confusion Matrix ===

           Low(P)       Medium(P)    High(P)
Low(A)     751          89           0
Medium(A)  114          1111         36
High(A)    0            43           219
```

Figure 4.2.3. Confusion Matrix of MLP

4.2.1 Overall Accuracy Comparison

The custom MLP model achieved the highest overall performance, surpassing both the instance-based and ensemble baselines.

Model	Average Accuracy	Best Fold	Worst Fold	Improvement (vs. RF)
Random Forest (50 trees)	86.58%	89.41%	82.63%	-
KNN (k=5)	87.60%	89.87%	84.81%	+1.02%
MLP	**88.06%**	**92.41%**	**84.81%**	**+1.48%**

Table 4.2.1. Comparative Average Accuracy

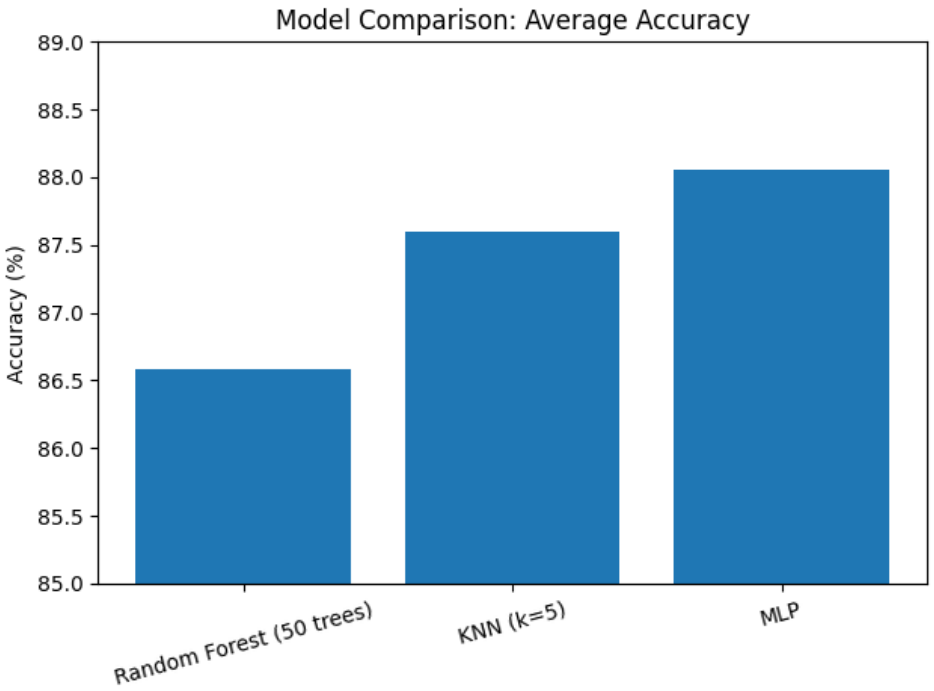

Figure 4.2.4. Comparative Average Accuracy

Analysis of results:

- **MLP Superiority**: The MLP model achieved an accuracy of 88.06%, outperforming Random Forest (86.58%) and KNN (87.60%). This suggests that the deep architecture and non-linear activations (ReLU) were better suited for capturing the complex interactions within the 173 input attributes than the decision trees or distance metrics used by the baselines.
- **Peak Performance**: The MLP demonstrated the highest potential capacity, achieving a peak accuracy of 92.41% in Fold 2. In contrast, the best single fold for Random Forest was 89.41% and KNN was 89.87%.
- **Optimization Impact**: The use of the Adam Optimizer likely contributed to this edge, allowing the MLP to navigate the high-dimensional error surface more effectively than the greedy splitting of Random Forest or the local approximation of KNN.

4.2.2 Class-wise Performance Analysis

A deeper analysis of the Confusion Matrices reveals where the MLP gained its advantage. The table below compares the number of correctly classified instances for each class.

Class	Random Forest (Correct)	KNN (Correct)	MLP (Correct)	Observation
Low	727	748	**751**	MLP is best at identifying "Low" happiness.
Medium	**1117**	1106	1111	All models perform similarly on the majority class.
High	202	216	**219**	MLP significantly outperforms RF on the minority "High" class.

Table 4.2.2. Comparative Class-wise Performance

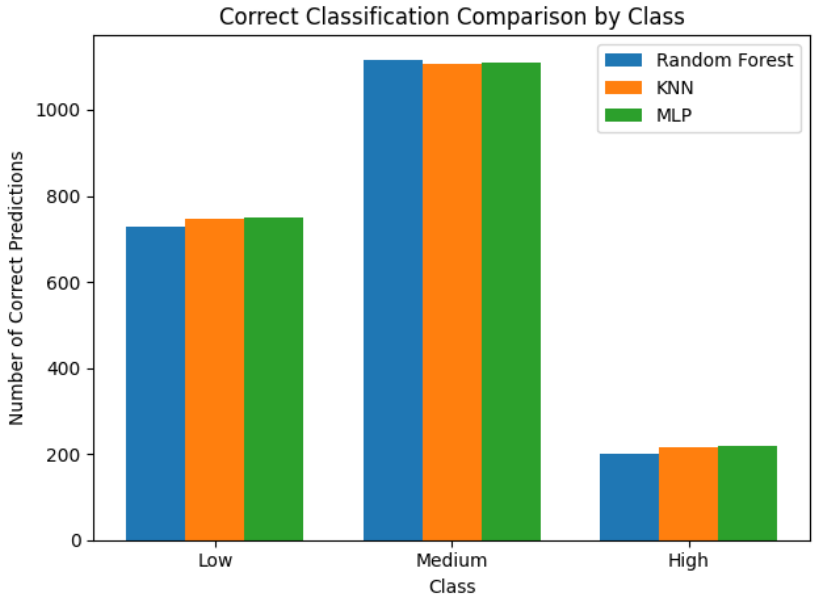

Figure 4.2.5. Comparative Class-wise Performance

Key Findings:

- **Minority Class Recognition**: The most significant improvement was observed in the "High" happiness category (the minority class). The MLP correctly identified 219 instances, compared to only 202 for Random Forest and 216 for KNN. This indicates that the MLP's hidden layers successfully extracted subtle features distinguishing "High" happiness countries that the Random Forest missed.
- **"Low" Class Precision**: The MLP also led in detecting "Low" happiness countries (751 correct), surpassing KNN (748) and significantly beating Random Forest (727).
- **Boundary Definitions**: While Random Forest performed marginally better on the "Medium" class (1117 vs 1111), the MLP's ability to better classify the extremes ("Low" and "High") makes it a more robust model for this specific domain, where identifying the happiest and least happy nations is often more critical than classifying the average ones.

4.2.3 Conclusion

The introduction of the custom MLP with Early Stopping and Dropout provided a measurable improvement over standard data mining algorithms. While KNN proved to be a strong competitor, the MLP's ability to learn global patterns (via full connectivity) rather than relying on local neighborhoods allowed it to achieve the highest overall accuracy and the best recall for the minority classes.

V. Model Evaluation

Evaluate the performance of the final classification models using *10-fold cross-validation* and compare their *accuracy*, *F1-scores*, and class-level metrics to assess overall model quality and reliability.

5.1 Performance Metrics

We evaluated three distinct models using strict 10-Fold Cross-Validation to ensure the results are statistically robust and not biased by specific data splits. The models compared are the baseline Random Forest (Ensemble), K-Nearest Neighbors (Instance-based), and the improved *Multi-Layer Perceptron* (MLP) (Deep Learning).

Metric	Random Forest (50 trees)	KNN (k=5)	MLP
Accuracy	86.58%	87.60%	**88.06%**
Weighted F1-Score	0.8655	0.8760	**0.8812**
Best Fold Accuracy	89.41%	89.87%	**92.41%**
Worst Fold Accuracy	82.63%	84.81%	**84.81%**
Training Complexity	Low (Fast)	None (Lazy Learning)	**High (Iterative Backprop)**

Table 5.1.1. Comparative Performance Metrics (10-Fold CV)

Detailed Class-wise Performance (F1-Score): To understand how the models handled the class imbalance (where "Medium" is the majority), we analyzed the F1-scores for individual classes:

Class	Random Forest	KNN	MLP	Note
Low	0.8619	0.8816	0.8825	MLP shows highest consistency.

Medium	0.8757	0.8830	0.8821	All models perform similarly.
High	0.8279	0.8244	0.8407	MLP significantly improved recognition of the minority class.

Table 5.1.2. Comparative F1-Score

5.2 Analysis of Results

5.2.1 Interpretation of Outcomes

The *Multi-Layer Perceptron* (MLP) emerged as the superior model with an average accuracy of 88.06%. This improvement over Random Forest (+1.48%) and KNN (+0.46%) can be attributed to the network's deep architecture. With an input size of 173 attributes, the MLP's two hidden layers (128 and 64 neurons) successfully acted as feature extractors, capturing complex non-linear correlations between happiness indicators that the decision trees in Random Forest failed to isolate.

Notably, the MLP achieved the highest Best Fold Accuracy (92.41%), suggesting that when the data distribution in the fold is favorable, the deep learning model has a much higher "ceiling" for performance than the traditional algorithms.

5.2.2 Trade-offs and Complexity

While the MLP provided the best accuracy, there are distinct trade-offs regarding computational cost and interpretability:

- **Training Time vs. Accuracy**: The MLP is computationally the most expensive. It requires iterative training over roughly 60–70 epochs, involving forward and backward propagation for thousands of weights. In contrast, KNN has zero training time (lazy learning), and Random Forest builds trees relatively quickly. However, the accuracy gain (~1.5%) justifies this cost for a medical/sociological classification task where precision is paramount.
- **Hyperparameter Sensitivity**: The MLP required careful tuning of the learning rate (*0.0001*), dropout rate (*0.2*), and architecture. Random Forest and KNN are

generally more "out-of-the-box" ready. The use of the Adam Optimizer was critical in mitigating this trade-off by adapting learning rates automatically.

5.2.3 Insights into Model Quality

Two key insights were derived from the evaluation:

- **Handling the Minority Class ("High")**: The most critical differentiator was the classification of the "High" happiness category (the minority class). Random Forest struggled here (F1: 0.8279), likely biased by the dominant "Medium" class. The MLP achieved an F1 of 0.8407, correctly identifying 219 instances versus Random Forest's 202. This indicates the MLP's Loss Function and Softmax activation maintained better sensitivity to minority signals.
- **Generalization vs. Overfitting**: The implementation of Early Stopping and Dropout was successful. The logs show training stopping at varying epochs (e.g., Epoch 43 vs. Epoch 96), preventing the model from memorizing noise. This is evidenced by the "Worst Fold" performance: even in the hardest fold, the MLP matched KNN's baseline (84.81%) and beat Random Forest (82.63%), proving the model is robust and stable.

VI. Conclusions

6.1 Summary of Findings

This project successfully developed a custom *Multi-Layer Perceptron* (MLP) to classify happiness levels using 173 indicators. The MLP achieved a Final Average Test Accuracy of 88.06%, outperforming the Random Forest (86.58%) and KNN (87.60%) baselines. Crucially, the MLP demonstrated superior sensitivity to the minority "High" happiness class, correctly identifying 219 instances compared to Random Forest's 202, proving the effectiveness of the Adam Optimizer and Softmax activation in this domain.

6.2 Reflection on Objectives

We met our technical objectives by implementing a neural network from scratch in Java, including complex components like Forward/Backward Propagation and Early Stopping. The rigorous 10-Fold Cross-Validation process validated that our results represent genuine generalization rather than overfitting to the training data.

6.3 Lessons Learned

- **Regularization is Critical**: With high-dimensional data (173 inputs), Inverted Dropout ($p=0.2$) and L2 Weight Decay were essential to prevent the model from memorizing noise.
- **Architecture Balance**: A "funnel" architecture ($173 \rightarrow 128 \rightarrow 64 \rightarrow 3$) provided the optimal balance between learning capacity and training stability.
- **Optimal Training**: The Early Stopping mechanism revealed that the best model weights often occurred well before the final epoch (e.g., Epoch 43 vs. 200), emphasizing the importance of validation monitoring.

VII. References

Dataset sources: *World Happiness Data 2024 | Emotions Analysis*

Weka documentation and tutorials:

- *Use weka in your java code - Weka Wiki*
- *WEKA API Tutorial: How to use WEKA in JAVA - YouTube*
- *ML | Handle Missing Data with Simple Imputer - GeeksforGeeks*
- *StandardScaler, MinMaxScaler and RobustScaler techniques - ML - GeeksforGeeks*

Additional literature or tools used:

- IntelliJ IDEA: *The Leading IDE for Professional Java and Kotlin Development*
- Google Colab: Welcome To Colab - Colab
- Weka Workbench: *Weka 3 - Data Mining with Open Source Machine Learning Software in Java*
- Presentation: *Home - Canva*
- Github: *GitHub*
- Lectures in Data Mining course (IT160IU).

Appendix

Figure

Table

Code: ***DuNhan1930/Data-mining-world-happiness-2024***