ASBD Project

Dataset: Rainfall Dataset

Group No: 5

Group Members:
Apurba Roy (ced19i053)
Nitish Kumar Singh (ced19i058)
Nitin koppera (ced19i001)

- To generate a target attribute with yes or no based on the provided rainfall dataset, we can use a threshold to divide the dataset into two classes. In this example, we will create a target attribute where 'yes' indicates that the annual rainfall is above the median value of the dataset, and 'no' indicates that the annual rainfall is below or equal to the median value.
- In this code, we are first loading the rainfall dataset and removing unnecessary columns. We then remove any rows with missing values and aggregate the data at a yearly level by summing the rainfall across all stations for each year.
- We then calculate the median rainfall for the entire dataset and use this value to create a new target attribute called 'target'. If the annual rainfall for a particular year is above the median, we set the value of 'target' to 'yes', otherwise, we set it to 'no'.

```
[ ] import pandas as pd

    # Load the dataset
    data = pd.read_csv('rainfall in India 1901-2015.csv')

    # Remove unnecessary columns
    data = data.drop(['SUBDIVISION', 'Jan-Feb', 'Mar-May', 'Jun-Sep', 'Oct-Dec'], axis=1)

    # Remove rows with missing values
    data = data.dropna()

    # Aggregate the data at a yearly level
    data = data.groupby('YEAR').sum()

    # Calculate the median rainfall
    median_rainfall = data['ANNUAL'].median()

    # Create a new column for the target attribute
    data['Target'] = data['ANNUAL'].apply(lambda x: 'yes' if x > median_rainfall else 'no')

    # Print the first five rows of the dataset, including the target attribute
    print(data.head(5))
```

```
      JAN  FEB  MAR  APR  MAY  JUN  JUL  \
YEAR
1901  0.1  0.3  0.6  7.6  8.1  9.1  9.1
1902  2.6  1.2  7.5  1.3  0.0  2.4  5.6
1903  0.0  0.3  0.3  0.0  2.1  6.4  8.1
1904  0.1  0.0  0.0  0.0  0.1  5.9  8.5
1905  0.0  0.0  1.1  3.2  4.3  8.3  9.1

      AUG  SEP  OCT  NOV  DEC  ANNUAL  Target
YEAR
1901  5.1  2.1  0.4  0.1  0.0  39.9     yes
1902  3.1  4.3  2.5  0.3  0.0  28.3      no
1903  4.8  2.4  0.1  0.0  0.0  20.5      no
1904  6.2  3.1  0.0  0.0  0.0  20.8      no
1905  5.5  2.5  0.0  0.5  0.0  26.1     yes
```

```
[ ] from sklearn.dataset import load_iris
    from sklearn.model_selection import train_test_split
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.metrics import accuracy_score

    X_train, X_test, y_train, y_test = train_test_split(data.drop('target', axis=1), data['target'], test_size=0.3, random_state=42)

    clf = DecisionTreeClassifier()
    clf.fit(X_train, y_train)

    # Fit the model to the training data
    tree.fit(X_train, y_train)

    # Predict the test set labels
    y_pred = tree.predict(X_test)

    # Compute the accuracy of the model
    accuracy = accuracy_score(y_test, y_pred)

    print("Accuracy:", accuracy)
```

```
Accuracy: 1.0
/usr/local/lib/python3.10/dist-packages/sphinx/ext/jsmath/__init__.pkg:178: DeprecationWarning: 'imp.reload_module' will not call 'transform_sys' automatically in the future. Please pass the new mod
    exec_module(sys.modules)
```

Naïve Bayes Classifier

```
[ ] import pandas as pd
    from sklearn.model_selection import train_test_split
    from sklearn.naive_bayes import GaussianNB
    from sklearn.metrics import accuracy_score

    # Load the dataset
    data = pd.read_csv('rainfall in India 1901-2015.csv')

    # Remove unnecessary columns
    data = data.drop(['SUBDIVISION', 'Jan-Feb', 'Mar-May', 'Jun-Sep', 'Oct-Dec'], axis=1)

    # Remove rows with missing values
    data = data.dropna()

    # Aggregate the data at a yearly level
    data = data.groupby('YEAR').sum()

    # Calculate the median rainfall
    median_rainfall = data['ANNUAL'].median()

    # Create a new column for the target attribute
    data['Target'] = data['ANNUAL'].apply(lambda x: 'yes' if x > median_rainfall else 'no')

    # Split the dataset into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(data.drop('Target', axis=1), data['Target'], test_size=0.3, random_state=42)

    # Create a Naive Bayes classifier
    clf = GaussianNB()
    clf.fit(X_train, y_train)

    # Make predictions on the testing set
    y_pred = clf.predict(X_test)

    # Calculate the performance of the classifier
    accuracy = accuracy_score(y_test, y_pred)
    print("Accuracy:", accuracy)
```

```
Accuracy: 0.8
/usr/local/lib/python3.10/dist-packages/sphinx/ext/jsmath/__init__.pkg:178: DeprecationWarning: 'imp.reload_module' will not call 'transform_sys' automatically in the future. Please pass the new mod
    exec_module(sys.modules)
```

Neural Network Classifier (Back Propagation Network(BPN))

- To generate a target attribute with yes or no based on the provided rainfall dataset, we can use a threshold to divide the dataset into two classes. In this example, we will create a target attribute where 'yes' indicates that the annual rainfall is above the median value of the dataset, and 'no' indicates that the annual rainfall is below or equal to the median value.
- In this code, we are first loading the rainfall dataset and removing unnecessary columns. We then remove any rows with missing values and aggregate the data at a yearly level by summing the rainfall across all stations for each year.
- We then calculate the median rainfall for the entire dataset and use this value to create a new target attribute called 'target'. If the annual rainfall for a particular year is above the median, we set the value of 'target' to 'yes', otherwise, we set it to 'no'.

```
[ ] import pandas as pd

    # Load the dataset
    data = pd.read_csv('rainfall in India 1901-2015.csv')

    # Remove unnecessary columns
    data = data.drop(['SUBDIVISION', 'Jan-Feb', 'Mar-May', 'Jun-Sep', 'Oct-Dec'], axis=1)

    # Remove rows with missing values
    data = data.dropna()

    # Aggregate the data at a yearly level
    data = data.groupby('YEAR').sum()

    # Calculate the median rainfall
    median_rainfall = data['ANNUAL'].median()

    # Create a new column for the target attribute
    data['Target'] = data['ANNUAL'].apply(lambda x: 'yes' if x > median_rainfall else 'no')

    # Print the first five rows of the dataset, including the target attribute
    print(data.head(5))
```

```
      JAN  FEB  MAR  APR  MAY  JUN  JUL  \
YEAR
1901  0.1  0.3  0.6  7.6  8.1  9.1  9.1
1902  2.6  1.2  7.5  1.3  0.0  2.4  5.6
1903  0.0  0.3  0.3  0.0  2.1  6.4  8.1
1904  0.1  0.0  0.0  0.0  0.1  5.9  8.5
1905  0.0  0.0  1.1  3.2  4.3  8.3  9.1

      AUG  SEP  OCT  NOV  DEC  ANNUAL  Target
YEAR
1901  5.1  2.1  0.4  0.1  0.0  39.9     yes
1902  3.1  4.3  2.5  0.3  0.0  28.3      no
1903  4.8  2.4  0.1  0.0  0.0  20.5      no
1904  6.2  3.1  0.0  0.0  0.0  20.8      no
1905  5.5  2.5  0.0  0.5  0.0  26.1     yes
```

```
[ ] import pandas as pd
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import StandardScaler

    # Load the aggregated dataset
    # data = pd.read_csv('aggregated_data.csv')

    # Split the data into a training set and a test set
    X_train, X_test, y_train, y_test = train_test_split(data.drop('Target', axis=1), data['Target'], test_size=0.3, random_state=42)

    # Normalize the data
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    # Create input and output vectors for the neural network
    X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
    X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)
    y_train = y_train.reshape(-1, 1)
    y_test = y_test.reshape(-1, 1)

    # Define the neural network architecture
    from keras.models import Sequential
    from keras.layers import Dense, Flatten

    model = Sequential()
    model.add(Flatten(input_shape=(X_train.shape[1], 1)))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(1, activation='linear'))

    # Compile the model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    # Train the model
    model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))
```

```
Epoch 1/10
113/113 [==============================] - 1s 12ms/step - loss: 0.7080 - accuracy: 0.5475 - val_loss: 0.6931 - val_accuracy: 0.5357
Epoch 2/10
113/113 [==============================] - 1s 13ms/step - loss: 0.6568 - accuracy: 0.6067 - val_loss: 0.6581 - val_accuracy: 0.6122
Epoch 3/10
113/113 [==============================] - 1s 12ms/step - loss: 0.6319 - accuracy: 0.6344 - val_loss: 0.6861 - val_accuracy: 0.5612
Epoch 4/10
113/113 [==============================] - 1s 13ms/step - loss: 0.5765 - accuracy: 0.6957 - val_loss: 0.7670 - val_accuracy: 0.4898
Epoch 5/10
113/113 [==============================] - 1s 13ms/step - loss: 0.5367 - accuracy: 0.7355 - val_loss: 0.7332 - val_accuracy: 0.4694
Epoch 6/10
113/113 [==============================] - 1s 13ms/step - loss: 0.5083 - accuracy: 0.7567 - val_loss: 0.7486 - val_accuracy: 0.4694
Epoch 7/10
113/113 [==============================] - 1s 13ms/step - loss: 0.4777 - accuracy: 0.7838 - val_loss: 0.8439 - val_accuracy: 0.4541
Epoch 8/10
113/113 [==============================] - 1s 13ms/step - loss: 0.4602 - accuracy: 0.7947 - val_loss: 0.8653 - val_accuracy: 0.4439
Epoch 9/10
113/113 [==============================] - 1s 13ms/step - loss: 0.4265 - accuracy: 0.8174 - val_loss: 0.9380 - val_accuracy: 0.5150
Epoch 10/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3977 - accuracy: 0.8424 - val_loss: 0.9598 - val_accuracy: 0.5102
Epoch 1/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3750 - accuracy: 0.8563 - val_loss: 1.0000 - val_accuracy: 0.5306
Epoch 2/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3576 - accuracy: 0.8676 - val_loss: 0.9639 - val_accuracy: 0.5204
Epoch 3/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3393 - accuracy: 0.8753 - val_loss: 0.9765 - val_accuracy: 0.5204
Epoch 4/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3229 - accuracy: 0.8782 - val_loss: 1.0158 - val_accuracy: 0.5102
Epoch 5/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3080 - accuracy: 0.8865 - val_loss: 0.9199 - val_accuracy: 0.5816
Epoch 6/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2932 - accuracy: 0.8943 - val_loss: 0.9805 - val_accuracy: 0.5204
Epoch 7/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2800 - accuracy: 0.8995 - val_loss: 1.0275 - val_accuracy: 0.4898
Epoch 8/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2671 - accuracy: 0.9047 - val_loss: 1.0471 - val_accuracy: 0.4898
Epoch 9/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2551 - accuracy: 0.9091 - val_loss: 0.9791 - val_accuracy: 0.5816
Epoch 10/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2446 - accuracy: 0.9091 - val_loss: 0.9715 - val_accuracy: 0.6020
Epoch 1/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2347 - accuracy: 0.9147 - val_loss: 1.0000 - val_accuracy: 0.5918
Epoch 2/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2249 - accuracy: 0.9169 - val_loss: 1.0209 - val_accuracy: 0.5867
Epoch 3/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2163 - accuracy: 0.9192 - val_loss: 0.9767 - val_accuracy: 0.6020
Epoch 4/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2067 - accuracy: 0.9214 - val_loss: 1.0253 - val_accuracy: 0.5918
Epoch 5/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1998 - accuracy: 0.9247 - val_loss: 1.0480 - val_accuracy: 0.6020
Epoch 6/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1919 - accuracy: 0.9270 - val_loss: 1.0154 - val_accuracy: 0.6020
Epoch 7/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1847 - accuracy: 0.9292 - val_loss: 1.0197 - val_accuracy: 0.6020
Epoch 8/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1788 - accuracy: 0.9315 - val_loss: 1.0480 - val_accuracy: 0.6020
Epoch 9/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1729 - accuracy: 0.9337 - val_loss: 1.0259 - val_accuracy: 0.6020
Epoch 10/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1677 - accuracy: 0.9371 - val_loss: 1.0375 - val_accuracy: 0.6020
Epoch 1/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1627 - accuracy: 0.9393 - val_loss: 1.0480 - val_accuracy: 0.6020
```

PySpark

```
[ ] !pip install pyspark
```

```
[ ] import pyspark
    from pyspark.sql import SparkSession
    from pyspark.sql.functions import *
    spark = SparkSession.builder.appname("mysql").getOrCreate()

    SparkSession - in-memory
    SparkContext
    Spark UI
    Version
        v3.4.0
    Master
        local[*]
    AppName
        PySpark
```

```
[ ] pip install findspark
```

```
[ ] from pyspark.sql.functions import col, when

    # Create a SparkSession
    spark = SparkSession.builder.appname("RainfallAnalysis").getOrCreate()

    # Load the dataset
    data = spark.read.format('csv').option('header', 'true').load('rainfall in India 1901-2015.csv')

    # Convert the "ANNUAL" column to a numeric data type
    data = data.withColumn('ANNUAL', col('ANNUAL').cast('double'))

    # Remove unnecessary columns
    data = data.drop('SUBDIVISION', 'Jan-Feb', 'Mar-May', 'Jun-Sep', 'Oct-Dec')

    # Remove rows with missing values
    data = data.dropna()

    # Add target column
    data = data.withColumn('target', when(col('ANNUAL') > 1000, 'yes').otherwise('no'))

    data.show(20)
```

```
+----+---+---+---+---+---+---+---+---+---+---+---+------+------+
|YEAR| JAN| FEB| MAR| APR| MAY| JUN| JUL| AUG| SEP| OCT| NOV| DEC|ANNUAL|Target|
+----+---+---+---+---+---+---+---+---+---+---+---+------+------+
|1901| 20.2|87.5|93.2|95.2|  78.5|179.5|185.5|165.1|152.6|102.5|68.5|19.6| 1267.3|  yes|
|1902|  0.1| 0.3| 0.6| 7.6|  8.1|  9.1|  9.1|  5.1|  2.1|  0.4| 0.1| 0.0|   39.9|  yes|
|1903|  2.6| 1.2| 7.5| 1.3|  0.0|  2.4|  5.6|  3.1|  4.3|  2.5| 0.3| 0.0|   28.3|   no|
|1904|  0.0| 0.3| 0.3| 0.0|  2.1|  6.4|  8.1|  4.8|  2.4|  0.1| 0.0| 0.0|   20.5|   no|
|1905|  0.1| 0.0| 0.0| 0.0|  0.1|  5.9|  8.5|  6.2|  3.1|  0.0| 0.0| 0.0|   20.8|   no|
|1906|  0.0| 0.0| 1.1| 3.2|  4.3|  8.3|  9.1|  5.5|  2.5|  0.0| 0.5| 0.0|   26.1|  yes|
|1907|  0.0| 0.0| 0.0| 0.0|  0.0|  7.8|  8.1|  4.8|  2.4|  0.1| 0.0| 0.0|   23.2|   no|
|1908|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1909|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1910|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1911|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1912|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1913|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1914|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1915|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1916|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1917|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1918|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1919|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
|1920|  0.0| 0.0| 0.0| 0.0|  0.0|  6.4|  8.5|  5.1|  2.1|  0.0| 0.0| 0.0|   22.1|   no|
+----+---+---+---+---+---+---+---+---+---+---+---+------+------+
only showing top 20 rows
```

```
[ ] import sklearn
    print(sklearn.__version__)
```

```
1.2.2
```

```
[ ] !pip install mlxtend
```

```
[ ] import findspark
    findspark.init()
    # Import the Spark session and the Spark DataFrame
    from pyspark.sql.dataframe import DataFrame
    # Create a Spark session
    spark = SparkSession.getOrCreate()

    # Split the data into training and testing sets
    train, test = df.randomSplit([0.7, 0.3])

    # Create a Spark DataFrame from the training and testing sets
    train_df = spark.createDataFrame(train)
    test_df = spark.createDataFrame(test)

    # Convert the DataFrames to numpy arrays
    train_array = train_df.toPandas().values
    test_array = test_df.toPandas().values

    # Create input and output vectors for the neural network
    X_train = train_array[:, 1:-1]
    y_train = train_array[:, -1]
    X_test = test_array[:, 1:-1]
    y_test = test_array[:, -1]

    # Define the neural network architecture
    from keras.models import Sequential
    from keras.layers import Dense, Flatten

    model = Sequential()
    model.add(Flatten(input_shape=(X_train.shape[1], 1)))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(1, activation='linear'))

    # Compile the model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    # Train the model
    model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))
```

```
Epoch 1/10
113/113 [==============================] - 1s 12ms/step - loss: 0.7080 - accuracy: 0.5475 - val_loss: 0.6931 - val_accuracy: 0.5357
Epoch 2/10
113/113 [==============================] - 1s 13ms/step - loss: 0.6568 - accuracy: 0.6067 - val_loss: 0.6581 - val_accuracy: 0.6122
Epoch 3/10
113/113 [==============================] - 1s 12ms/step - loss: 0.6319 - accuracy: 0.6344 - val_loss: 0.6861 - val_accuracy: 0.5612
Epoch 4/10
113/113 [==============================] - 1s 13ms/step - loss: 0.5765 - accuracy: 0.6957 - val_loss: 0.7670 - val_accuracy: 0.4898
Epoch 5/10
113/113 [==============================] - 1s 13ms/step - loss: 0.5367 - accuracy: 0.7355 - val_loss: 0.7332 - val_accuracy: 0.4694
Epoch 6/10
113/113 [==============================] - 1s 13ms/step - loss: 0.5083 - accuracy: 0.7567 - val_loss: 0.7486 - val_accuracy: 0.4694
Epoch 7/10
113/113 [==============================] - 1s 13ms/step - loss: 0.4777 - accuracy: 0.7838 - val_loss: 0.8439 - val_accuracy: 0.4541
Epoch 8/10
113/113 [==============================] - 1s 13ms/step - loss: 0.4602 - accuracy: 0.7947 - val_loss: 0.8653 - val_accuracy: 0.4439
Epoch 9/10
113/113 [==============================] - 1s 13ms/step - loss: 0.4265 - accuracy: 0.8174 - val_loss: 0.9380 - val_accuracy: 0.5150
Epoch 10/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3977 - accuracy: 0.8424 - val_loss: 0.9598 - val_accuracy: 0.5102
Epoch 1/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3750 - accuracy: 0.8563 - val_loss: 1.0000 - val_accuracy: 0.5306
Epoch 2/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3576 - accuracy: 0.8676 - val_loss: 0.9639 - val_accuracy: 0.5204
Epoch 3/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3393 - accuracy: 0.8753 - val_loss: 0.9765 - val_accuracy: 0.5204
Epoch 4/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3229 - accuracy: 0.8782 - val_loss: 1.0158 - val_accuracy: 0.5102
Epoch 5/10
113/113 [==============================] - 1s 13ms/step - loss: 0.3080 - accuracy: 0.8865 - val_loss: 0.9199 - val_accuracy: 0.5816
Epoch 6/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2932 - accuracy: 0.8943 - val_loss: 0.9805 - val_accuracy: 0.5204
Epoch 7/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2800 - accuracy: 0.8995 - val_loss: 1.0275 - val_accuracy: 0.4898
Epoch 8/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2671 - accuracy: 0.9047 - val_loss: 1.0471 - val_accuracy: 0.4898
Epoch 9/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2551 - accuracy: 0.9091 - val_loss: 0.9791 - val_accuracy: 0.5816
Epoch 10/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2446 - accuracy: 0.9091 - val_loss: 0.9715 - val_accuracy: 0.6020
Epoch 1/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2347 - accuracy: 0.9147 - val_loss: 1.0000 - val_accuracy: 0.5918
Epoch 2/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2249 - accuracy: 0.9169 - val_loss: 1.0209 - val_accuracy: 0.5867
Epoch 3/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2163 - accuracy: 0.9192 - val_loss: 0.9767 - val_accuracy: 0.6020
Epoch 4/10
113/113 [==============================] - 1s 13ms/step - loss: 0.2067 - accuracy: 0.9214 - val_loss: 1.0253 - val_accuracy: 0.5918
Epoch 5/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1998 - accuracy: 0.9247 - val_loss: 1.0480 - val_accuracy: 0.6020
Epoch 6/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1919 - accuracy: 0.9270 - val_loss: 1.0154 - val_accuracy: 0.6020
Epoch 7/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1847 - accuracy: 0.9292 - val_loss: 1.0197 - val_accuracy: 0.6020
Epoch 8/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1788 - accuracy: 0.9315 - val_loss: 1.0480 - val_accuracy: 0.6020
Epoch 9/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1729 - accuracy: 0.9337 - val_loss: 1.0259 - val_accuracy: 0.6020
Epoch 10/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1677 - accuracy: 0.9371 - val_loss: 1.0375 - val_accuracy: 0.6020
Epoch 1/10
113/113 [==============================] - 1s 13ms/step - loss: 0.1627 - accuracy: 0.9393 - val_loss: 1.0480 - val_accuracy: 0.6020
```

Cómo pedir productos · Cancel contracts here