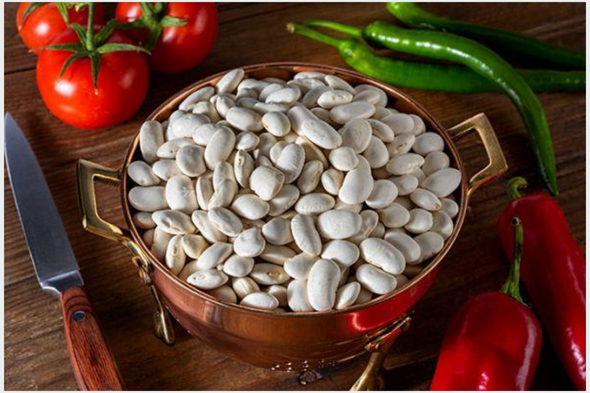
Dry Bean Classification

A high-dimensional multiclass classification project

A Deep Learning Approach



Introduction

Background:

- The global agricultural sector is continually evolving and expanding.
- Quality control and classification of crops are key aspects in this process.
- Dry beans present a unique classification challenge due to similarities among different types.

Importance:

- Accurate classification directly impacts the market status of different bean varieties.
- Enhancing precision and efficiency in seed classification promotes uniform seed distribution.

Problem Statement

- **Objective:** Our goal was to create a high-dimensional, multiclass classification model. This model aims to accurately identify seven distinct types of dry beans.
- **Approach:** We've combined supervised learning with XGBoost and advanced Artificial Neural Networks (ANN). This approach provides an innovative solution to an intricate agricultural challenge.
- **Evaluation:** The success of our project is measured by the accuracy and precision of bean classification. Our model's efficacy will also be gauged based on its generalizability and ability to scale up.

Dataset

- Available on the UC Irvine Machine Learning Repository
- **Name**: Dry Bean Dataset
- **Creator**: KOKLU, M. and OZKAN, I.A., (2020)
- **DOI**: https://doi.org/10.1016/j.compag.2020.105507
- **Link**: https://archive.ics.uci.edu/dataset/602/dry+bean+dataset

Data Overview

```
[ ] df.head(5)
```

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexArea	EquivDiameter	Extent	Solidity	roundness	Compactness	ShapeFactor1	ShapeFactor2	ShapeFactor3	ShapeFactor4	Class
0	28395	610.291	208.178117	173.888747	1.197191	0.549812	28715	190.141097	0.763923	0.988856	0.958027	0.913358	0.007332	0.003147	0.834222	0.998724	SEKER
1	28734	638.018	200.524796	182.734419	1.097356	0.411785	29172	191.272751	0.783968	0.984986	0.887034	0.953861	0.006979	0.003564	0.909851	0.998430	SEKER
2	29380	624.110	212.826130	175.931143	1.209713	0.562727	29690	193.410904	0.778113	0.989559	0.947849	0.908774	0.007244	0.003048	0.825871	0.999066	SEKER
3	30008	645.884	210.557999	182.516516	1.153638	0.498616	30724	195.467062	0.782681	0.976696	0.903936	0.928329	0.007017	0.003215	0.861794	0.994199	SEKER
4	30140	620.134	201.847882	190.279279	1.060798	0.333680	30417	195.896503	0.773098	0.990893	0.984877	0.970516	0.006697	0.003665	0.941900	0.999166	SEKER

Data Overview

```
df.info()
```

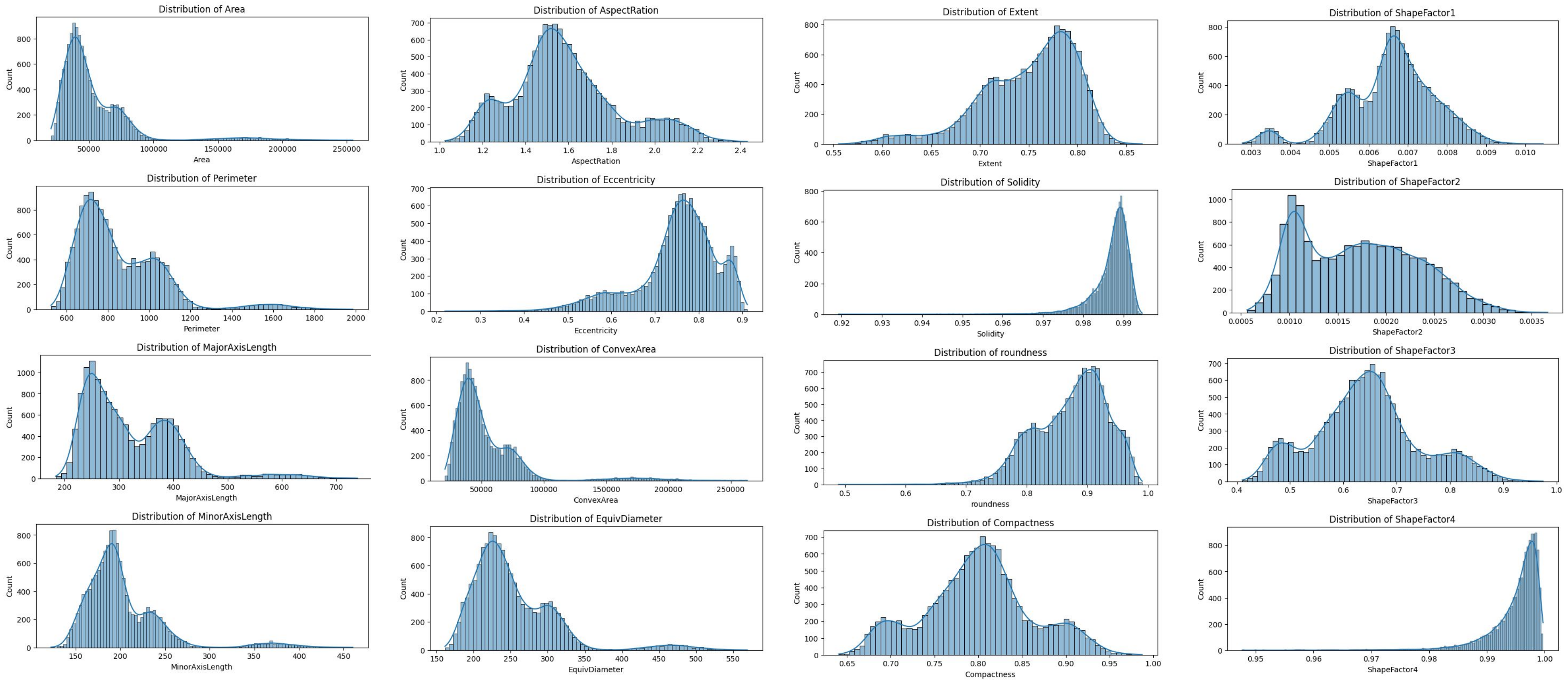
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13611 entries, 0 to 13610
Data columns (total 17 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Area             13611 non-null  int64
 1   Perimeter        13611 non-null  float64
 2   MajorAxisLength  13611 non-null  float64
 3   MinorAxisLength  13611 non-null  float64
 4   AspectRation     13611 non-null  float64
 5   Eccentricity     13611 non-null  float64
 6   ConvexArea       13611 non-null  int64
 7   EquivDiameter    13611 non-null  float64
 8   Extent           13611 non-null  float64
 9   Solidity         13611 non-null  float64
 10  roundness        13611 non-null  float64
 11  Compactness      13611 non-null  float64
 12  ShapeFactor1     13611 non-null  float64
 13  ShapeFactor2     13611 non-null  float64
 14  ShapeFactor3     13611 non-null  float64
 15  ShapeFactor4     13611 non-null  float64
 16  Class            13611 non-null  object
dtypes: float64(14), int64(2), object(1)
memory usage: 1.8+ MB
```

Check to see if there is any missing data.

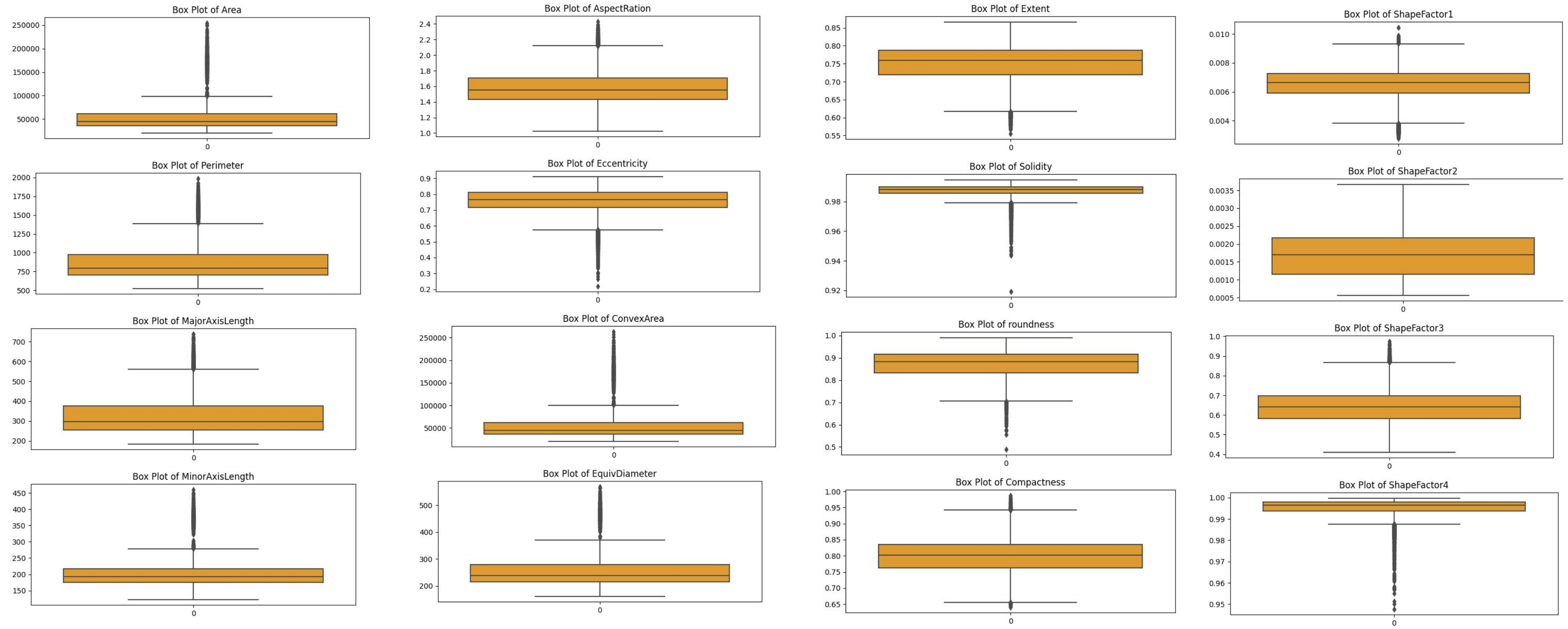
```
[ ] missing_data = df.isnull().sum()
    print(missing_data)
```

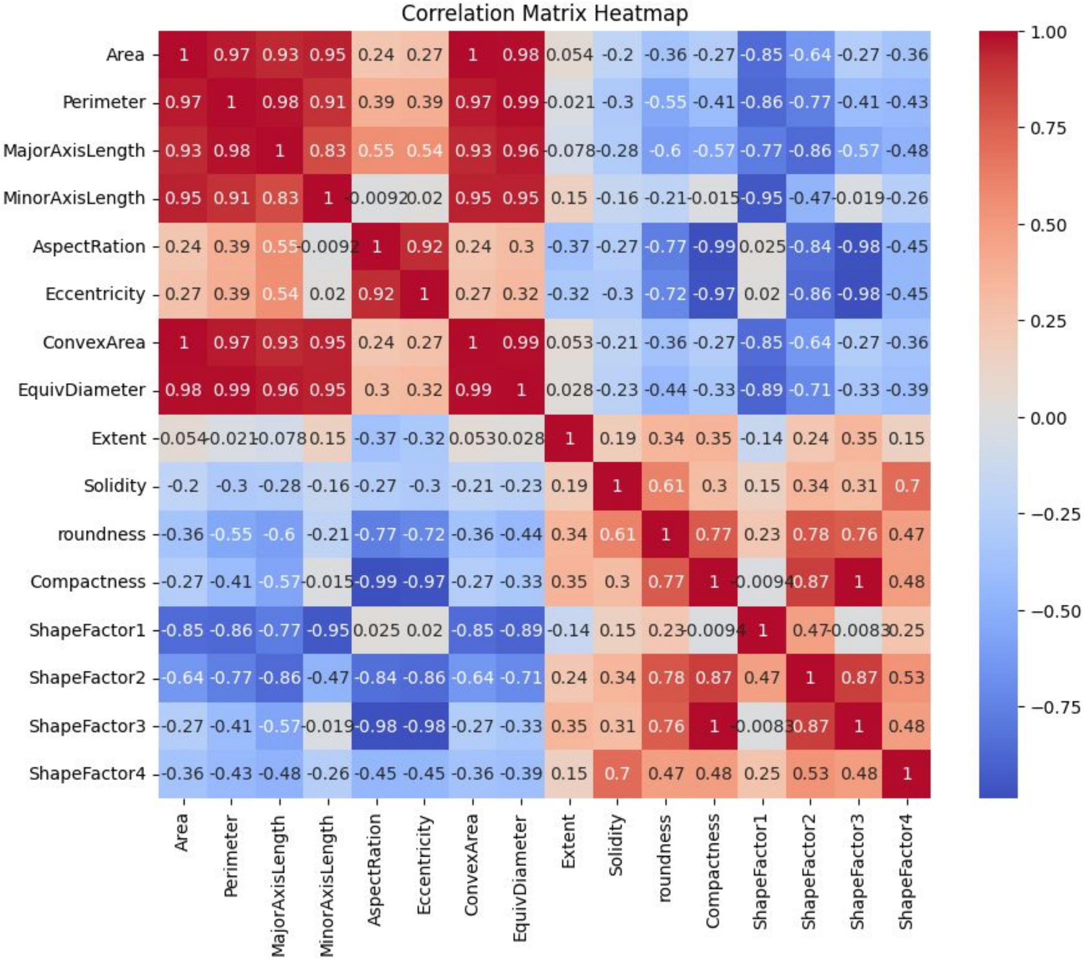
```
Area               0
Perimeter          0
MajorAxisLength    0
MinorAxisLength    0
AspectRation       0
Eccentricity       0
ConvexArea         0
EquivDiameter      0
Extent             0
Solidity           0
roundness          0
Compactness        0
ShapeFactor1       0
ShapeFactor2       0
ShapeFactor3       0
ShapeFactor4       0
Class              0
dtype: int64
```

Data Visualization

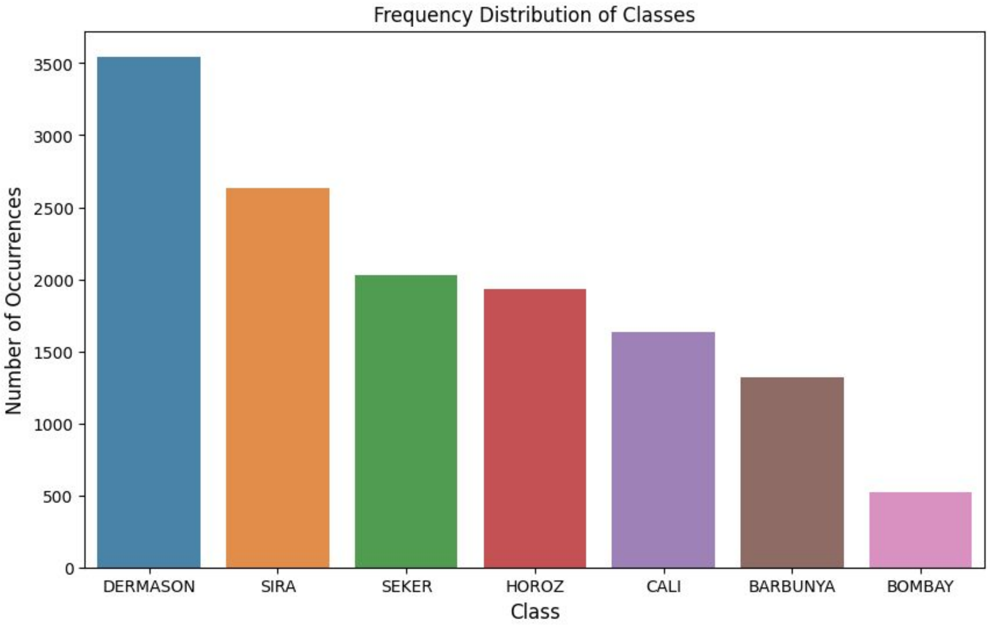


Data Visualization



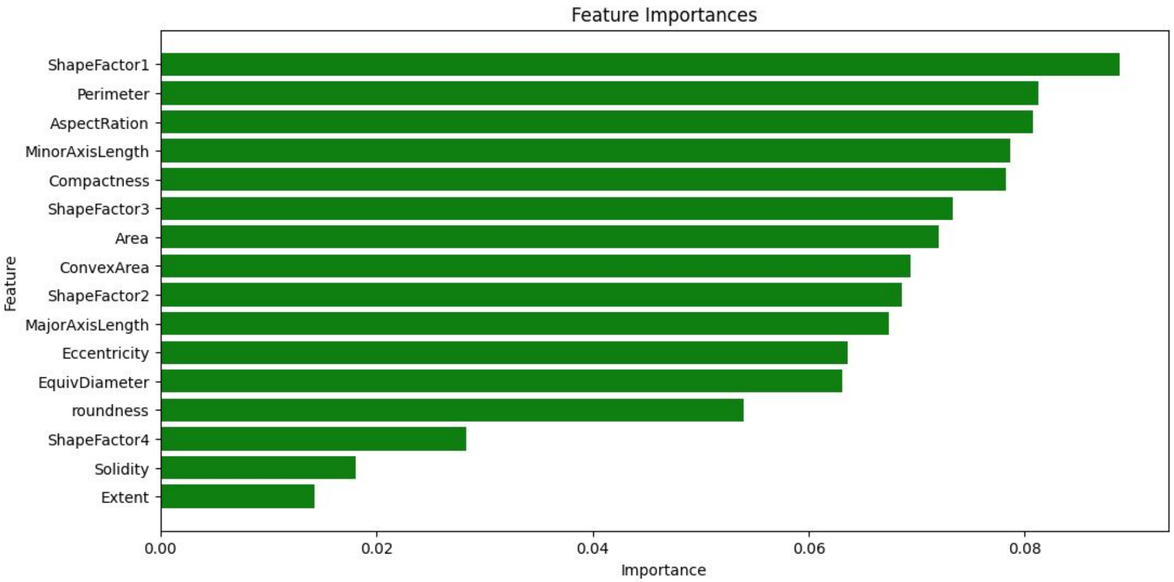

Correlation Matrix Heatmap
Area 1 0.97 0.93 0.95 0.24 0.27 1 0.98 0.054 -0.2 -0.36 -0.27 -0.85 -0.64 -0.27 -0.36
Perimeter 0.97 1 0.98 0.91 0.39 0.39 0.97 0.99 -0.021 -0.3 -0.55 -0.41 -0.86 -0.77 -0.41 -0.43
MajorAxisLength 0.93 0.98 1 0.83 0.55 0.54 0.93 0.96 -0.078 -0.28 -0.6 -0.57 -0.77 -0.86 -0.57 -0.48
MinorAxisLength 0.95 0.91 0.83 1 -0.0092 0.02 0.95 0.95 0.15 -0.16 -0.21 -0.015 -0.95 -0.47 -0.019 -0.26
AspectRation 0.24 0.39 0.55 -0.0092 1 0.92 0.24 0.3 -0.37 -0.27 -0.77 -0.99 0.025 -0.84 -0.98 -0.45
Eccentricity 0.27 0.39 0.54 0.02 0.92 1 0.27 0.32 -0.32 -0.3 -0.72 -0.97 0.02 -0.86 -0.98 -0.45
ConvexArea 1 0.97 0.93 0.95 0.24 0.27 1 0.99 0.053 -0.21 -0.36 -0.27 -0.85 -0.64 -0.27 -0.36
EquivDiameter 0.98 0.99 0.96 0.95 0.3 0.32 0.99 1 0.028 -0.23 -0.44 -0.33 -0.89 -0.71 -0.33 -0.39
Extent 0.054 -0.021 -0.078 0.15 -0.37 -0.32 0.053 0.028 1 0.19 0.34 0.35 -0.14 0.24 0.35 0.15
Solidity -0.2 -0.3 -0.28 -0.16 -0.27 -0.3 -0.21 -0.23 0.19 1 0.61 0.3 0.15 0.34 0.31 0.7
roundness -0.36 -0.55 -0.6 -0.21 -0.77 -0.72 -0.36 -0.44 0.34 0.61 1 0.77 0.23 0.78 0.76 0.47
Compactness -0.27 -0.41 -0.57 -0.015 -0.99 -0.97 -0.27 -0.33 0.35 0.3 0.77 1 -0.0094 0.87 1 0.48
ShapeFactor1 -0.85 -0.86 -0.77 -0.95 0.025 0.02 -0.85 -0.89 -0.14 0.15 0.23 -0.0094 1 0.47 -0.0083 0.25
ShapeFactor2 -0.64 -0.77 -0.86 -0.47 -0.84 -0.86 -0.64 -0.71 0.24 0.34 0.78 0.87 0.47 1 0.87 0.53
ShapeFactor3 -0.27 -0.41 -0.57 -0.019 -0.98 -0.98 -0.27 -0.33 0.35 0.31 0.76 1 -0.0083 0.87 1 0.48
ShapeFactor4 -0.36 -0.43 -0.48 -0.26 -0.45 -0.45 -0.36 -0.39 0.15 0.7 0.47 0.48 0.25 0.53 0.48 1
Area Perimeter MajorAxisLength MinorAxisLength AspectRation Eccentricity ConvexArea EquivDiameter Extent Solidity roundness Compactness ShapeFactor1 ShapeFactor2 ShapeFactor3 ShapeFactor4
1.00 0.75 0.50 0.25 0.00 −0.25 −0.50 −0.75

Data Visualization



- There is evidence for class imbalance
- Solution: oversampling with SMOTE

Feature Selection



- Use a tree classifier to extract each features' importance
- Sort them descendingly
- Eliminate the 3 least important features: Extent, Solidity, ShapeFactor4

Selected Features

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexArea	EquivDiameter	roundness	Compactness	ShapeFactor1	ShapeFactor2	ShapeFactor3
0	28395	610.291	208.178117	173.888747	1.197191	0.549812	28715	190.141097	0.958027	0.913358	0.007332	0.003147	0.834222
1	28734	638.018	200.524796	182.734419	1.097356	0.411785	29172	191.272751	0.887034	0.953861	0.006979	0.003564	0.909851
2	29380	624.110	212.826130	175.931143	1.209713	0.562727	29690	193.410904	0.947849	0.908774	0.007244	0.003048	0.825871
3	30008	645.884	210.557999	182.516516	1.153638	0.498616	30724	195.467062	0.903936	0.928329	0.007017	0.003215	0.861794
4	30140	620.134	201.847882	190.279279	1.060798	0.333680	30417	195.896503	0.984877	0.970516	0.006697	0.003665	0.941900

Artificial Neural Network (ANN)

The proposed procedure will be:

1. Data splitting into training (80%) and testing (20%) set
2. Data standardization using Standard Scaler
3. Solving class imbalance with oversampling using SMOTE
4. Label encoding to transform categorical labels into numerical labels
5. Model building and training
6. Model testing
7. Hyperparameters tuning using keras_tuner

1. Data splitting into training (80%) and testing (20%) set
2. Data standardization using Standard Scaler
3. Solving class imbalance with oversampling using SMOTE

```
[ ] # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

    # Standardize the features
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    # Apply SMOTE to oversample the minority class
    smote = SMOTE(random_state=1)
    X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
```

4. Label encoding to transform categorical labels into numerical labels

```
[ ] from sklearn.preprocessing import LabelEncoder

    encoder = LabelEncoder()
    y_train_resampled_encoded = encoder.fit_transform(y_train_resampled)
    y_test_encoded = encoder.transform(y_test)
```

```
[ ] # Build the ANN model
    model = Sequential()
    model.add(Dense(64, activation='relu', input_dim=X_train_resampled.shape[1]))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(10, activation='softmax'))

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

```
[ ] import time
    # Start the timer
    start_time = time.time()

    # Train the model
    model.fit(X_train_resampled, y_train_resampled_encoded, epochs=100, batch_size=32, verbose=1)

    # Stop the timer and calculate the runtime
    runtime = time.time() - start_time
```

```
Epoch 1/100
622/622 [==============================] - 2s 2ms/step - loss: 0.3288 - accuracy: 0.8904
Epoch 2/100
622/622 [==============================] - 2s 2ms/step - loss: 0.2020 - accuracy: 0.9274
Epoch 3/100
622/622 [==============================] - 2s 3ms/step - loss: 0.1954 - accuracy: 0.9299
Epoch 4/100
622/622 [==============================] - 2s 3ms/step - loss: 0.1929 - accuracy: 0.9297
Epoch 5/100
622/622 [==============================] - 1s 2ms/step - loss: 0.1896 - accuracy: 0.9320
Epoch 6/100
622/622 [==============================] - 1s 2ms/step - loss: 0.1888 - accuracy: 0.9314
Epoch 7/100
622/622 [==============================] - 1s 2ms/step - loss: 0.1856 - accuracy: 0.9318
Epoch 8/100
622/622 [==============================] - 1s 2ms/step - loss: 0.1848 - accuracy: 0.9320
Epoch 9/100
622/622 [==============================] - 1s 2ms/step - loss: 0.1805 - accuracy: 0.9336
Epoch 10/100
622/622 [==============================] - 1s 2ms/step - loss: 0.1796 - accuracy: 0.9344
Epoch 11/100
622/622 [==============================] - 1s 2ms/step - loss: 0.1799 - accuracy: 0.9333
Epoch 12/100
622/622 [==============================] - 1s 2ms/step - loss: 0.1765 - accuracy: 0.9343
Epoch 13/100
622/622 [==============================] - 1s 2ms/step - loss: 0.1766 - accuracy: 0.9339
```

Initial Results

```
[ ] # Make predictions
    y_pred_prob = model.predict(X_test)
    y_pred = y_pred_prob.argmax(axis=1)

    # Evaluate the model
    accuracy = accuracy_score(y_test_encoded, y_pred)
    report = classification_report(y_test_encoded, y_pred)

    print("Accuracy:", round(accuracy,4))
    print("Classification Report:\n", report)
    print("Runtime:", round(runtime, 2), "seconds")
```

```
86/86 [==============================] - 1s 3ms/step
Accuracy: 0.924
Classification Report:
               precision    recall  f1-score   support

           0       0.88      0.94      0.91       270
           1       1.00      1.00      1.00       103
           2       0.96      0.89      0.92       333
           3       0.91      0.94      0.92       705
           4       0.96      0.97      0.96       386
           5       0.93      0.97      0.95       405
           6       0.90      0.84      0.87       521

    accuracy                           0.92      2723
   macro avg       0.93      0.93      0.93      2723
weighted avg       0.92      0.92      0.92      2723

Runtime: 142.57 seconds
```

Hyperparameters Tuning

```
from keras_tuner import HyperModel, RandomSearch

# Define the model architecture within a function, using hyperparameters where desired
def build_model(hp):
    model = Sequential()
    model.add(Dense(hp.Int('nodes', min_value=32, max_value=512, step=32),
                    activation='relu',
                    input_dim=X_train_resampled.shape[1]))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(10, activation='softmax'))

    model.compile(
        optimizer=keras.optimizers.Adam(
            hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4])),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

    return model
```

Hyperparameters Tuning

```
# Define the tuner
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,  # set number of trials, in a real project this should be a higher number
    executions_per_trial=3,  # model will be trained this many times per trial to average out performance
    directory='.',
    project_name='keras_tuner_demo')

# Start the search for the best hyperparameters
tuner.search(X_train_resampled, y_train_resampled_encoded,
             validation_split=0.2,  # hold out 20% of the data for validation
             epochs=5)  # set number of epochs, in a real project this should be a higher number

# Get the optimal hyperparameters
best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]
```

```
Trial 5 Complete [00h 00m 33s]
val_accuracy: 0.9345567226409912

Best val_accuracy So Far: 0.935897429784139
Total elapsed time: 00h 02m 52s
```

Hyperparameters Tuning

```
# Print the optimal hyperparameters
print(f"""
The hyperparameter search is complete.
The optimal number of nodes in the first densely-connected layer is {best_hps.get('nodes')}
and the optimal learning rate for the optimizer is {best_hps.get('learning_rate')}.
""")
```

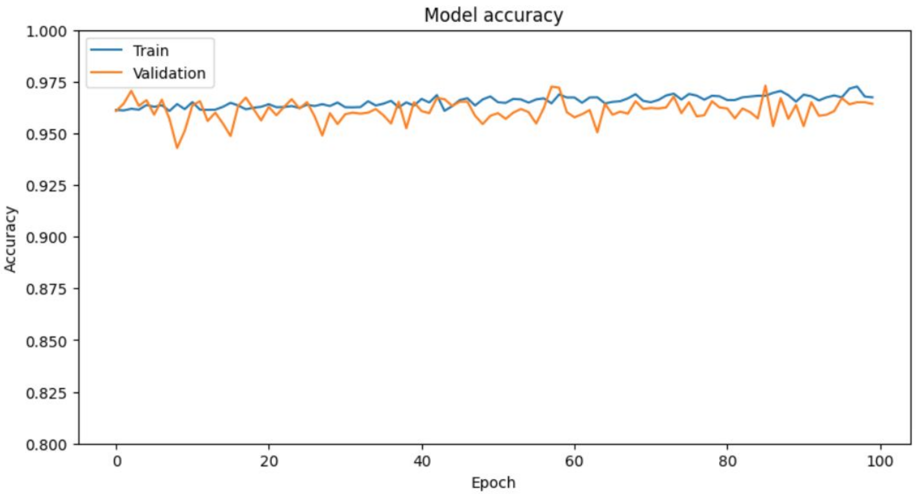
```
The hyperparameter search is complete.
The optimal number of nodes in the first densely-connected layer is 480
and the optimal learning rate for the optimizer is 0.001.
```

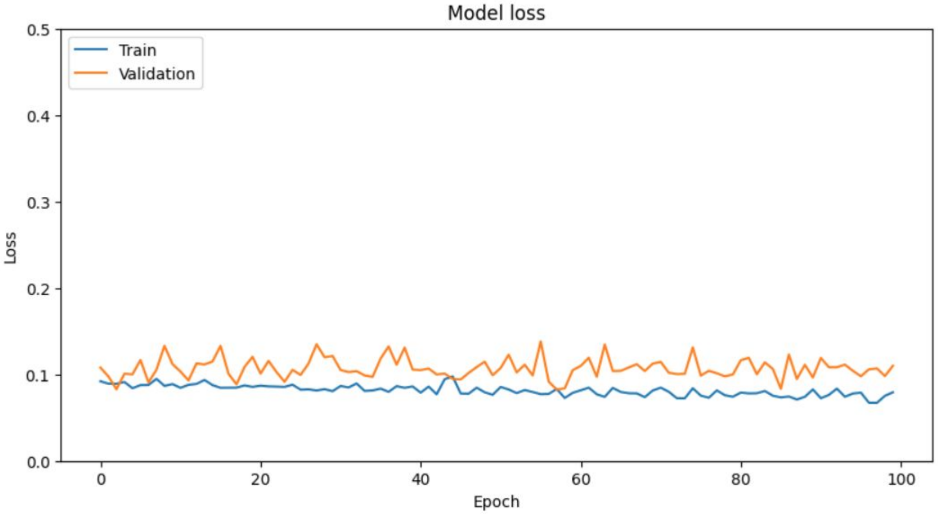
Training the Tuned Model

```
86/86 [==============================] - 0s 1ms/step
Accuracy: 0.9273
Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.89      0.91       270
           1       1.00      1.00      1.00       103
           2       0.90      0.94      0.92       333
           3       0.91      0.94      0.93       705
           4       0.96      0.97      0.96       386
           5       0.96      0.94      0.95       405
           6       0.89      0.86      0.88       521

    accuracy                           0.93      2723
   macro avg       0.94      0.93      0.94      2723
weighted avg       0.93      0.93      0.93      2723

Runtime: 178.75 seconds
```

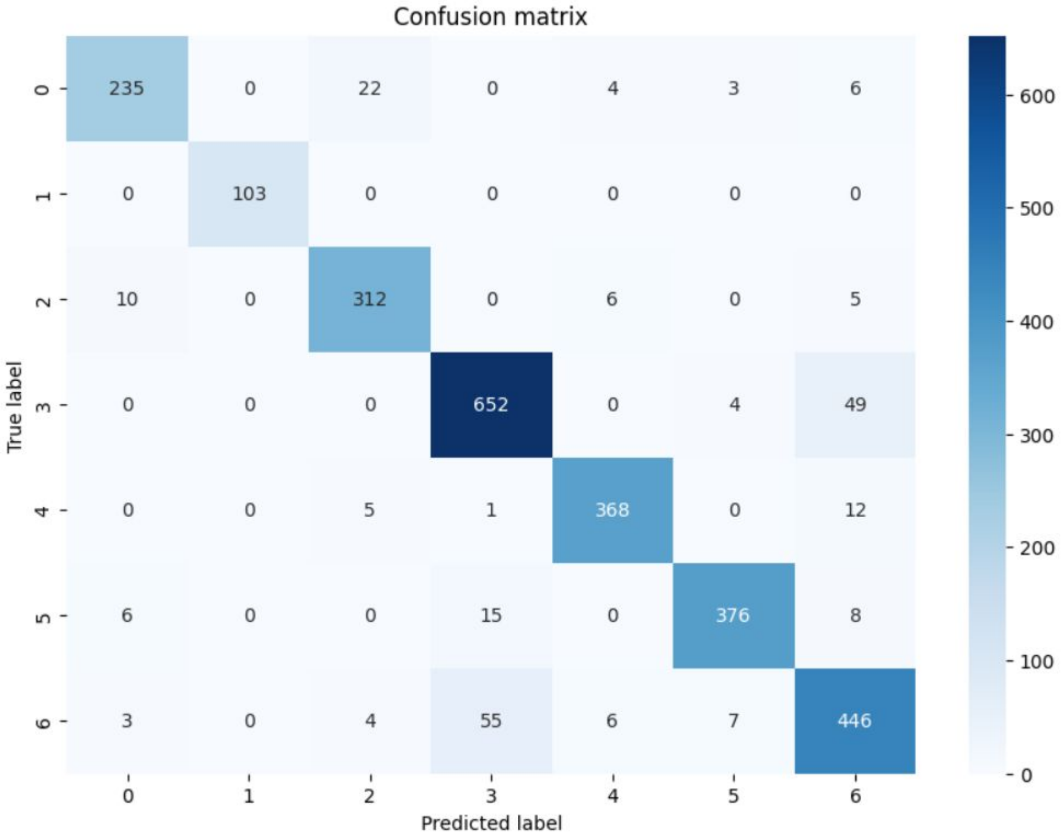
→ Marginal improvement in accuracy, yet significant increase in training runtime (143s to 179s)
→ It is better to keep the initial model, or further experiment with kera_tuner is needed.

Result Visualization





Result Visualization



XGBoost

```python
import time

# Create an XGBoost classifier
xgb_classifier = xgb.XGBClassifier(
    objective='multi:softprob',
    num_class=7,  # number of classes
    random_state=1)

# Start the timer
start_time = time.time()

# Train the XGBoost classifier
xgb_classifier.fit(X_train_resampled, y_train_resampled)

# Stop the timer and calculate the runtime
runtime = time.time() - start_time
```

```
Accuracy: 0.9207
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.88      0.90       270
           1       1.00      1.00      1.00       103
           2       0.92      0.92      0.92       333
           3       0.92      0.92      0.92       705
           4       0.96      0.96      0.96       386
           5       0.96      0.94      0.95       405
           6       0.85      0.88      0.87       521

    accuracy                           0.92      2723
   macro avg       0.93      0.93      0.93      2723
weighted avg       0.92      0.92      0.92      2723

Runtime: 39.68 seconds
```

Conclusion

Model	Training Accuracy	Testing Accuracy	Runtime
ANN	0.9508	0.924	142.57 s
ANN Tuned	0.9517	0.9273	178.75 s
XGBoost	0.9982	0.9207	39.68 s

Key Findings

- Deep learning architectures like ANN are advanced but don't always ensure better testing accuracy or efficiency.
- Our experiments revealed XGBoost's superior performance in terms of efficiency, running 4 times faster than ANN.
- The testing accuracy of the XGBoost model was comparable to the ANN model, achieving a score of over 0.92.

Insights

- Deep Learning architectures demonstrate their utility in specific cases like computer vision or processing in-depth language model transformers with large parameters.
- For high dimensional multiclass classification tasks with numerical features, it's recommended to initially test less resource-intensive supervised methods such as Random Forest, SVM, or XGBoost.

Recommendations

- The optimal approach balances accuracy for future generalizability and efficiency for large-scale production.
- Our findings suggest careful selection of methods based on specific use cases rather than a blanket preference for more advanced methods.