

QUESTION 1

HOUSE PRICE PREDICTION (LINEAR VS MLP)

1. Load and prepare data

The required libraries are imported, including **pandas** for data manipulation, **numpy** for numerical operations, **sklearn** for model selection, linear models, neural networks, and metrics, **matplotlib** for plotting, and **joblib** for saving models.

```
Question 1

[ ] 1 # =====
2 # HOUSE PRICE PREDICTION (LINEAR vs MLP)
3 # =====
4
5 import pandas as pd
6 import numpy as np
7 from sklearn.model_selection import train_test_split
8 from sklearn.linear_model import LinearRegression
9 from sklearn.neural_network import MLPRegressor
10 from sklearn.preprocessing import StandardScaler
11 from sklearn.metrics import mean_squared_error, r2_score
12 import matplotlib.pyplot as plt
13 import joblib
14
15 # =====
16 # 1. Load and prepare data
17 # =====
18 df = pd.read_csv('housing.csv')
19
20 # Select target and top 5 predictive features
21 target_col = 'price'
22 features = ['sqft_living', 'grade', 'sqft_above', 'sqft_living15', 'bathrooms']
23 df = df[features + [target_col]].dropna()
24
25 # Log-transform the target to stabilize variance
26 df['log_price'] = np.log(df['price'] + 1)
27
28 # Train-test split
29 X = df[features].values
```

2. Standardize features and Linear Regression (baseline performance)

The features are standardized using **StandardScaler** to normalize the data. A **Linear Regression** model is then trained as a baseline. The predictions and test target values are converted back from the log scale to the original scale for performance evaluation using **Root Mean Squared Error (RMSE)** and **R-squared (R^2)**.

```

30 y = df['log_price'].values
31 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
32
33 # Standardize features
34 scaler = StandardScaler()
35 X_train_s = scaler.fit_transform(X_train)
36 X_test_s = scaler.transform(X_test)
37
38 # =====
39 # 2. Linear Regression (baseline performance)
40 # =====
41 lr = LinearRegression()
42 lr.fit(X_train_s, y_train)
43 y_pred_lr_log = lr.predict(X_test_s)
44
45 # Convert back from log to original scale
46 y_pred_lr = np.exp(y_pred_lr_log) - 1
47 y_test_orig = np.exp(y_test) - 1
48
49 rmse_lr = np.sqrt(mean_squared_error(y_test_orig, y_pred_lr))
50 r2_lr = r2_score(y_test_orig, y_pred_lr)
51 print("LINEAR REGRESSION PERFORMANCE:")
52 print(f"  RMSE = {rmse_lr:.2f}")
53 print(f"  R²   = {r2_lr:.4f}")
54
55 # =====
56 # 3. Neural Network Regression (tuned)
57 # =====
58 mlp = MLPRegressor(
59     hidden_layer_sizes=(200, 100),
60     activation='tanh',

```

3. Neural Network Regression (tuned)

A **Multi-layer Perceptron (MLP) Regressor** (Neural Network) is initialized with two hidden layers (200 and 100 neurons), **tanh** activation, and the **adam** solver. The model is trained, and its performance is evaluated similarly to the Linear Regression model.

```

60     activation= 'tanh',
61     solver='adam',
62     alpha=1e-4,
63     learning_rate_init=1e-4,
64     early_stopping=True,
65     max_iter=800,
66     random_state=42
67 )
68
69 mlp.fit(X_train_s, y_train)
70 y_pred_mlp_log = mlp.predict(X_test_s)
71 y_pred_mlp = np.exp(y_pred_mlp_log) - 1
72
73 rmse_mlp = np.sqrt(mean_squared_error(y_test_orig, y_pred_mlp))
74 r2_mlp = r2_score(y_test_orig, y_pred_mlp)
75 print("\nNEURAL NETWORK (MLP) PERFORMANCE:")
76 print(f"   RMSE = {rmse_mlp:.2f}")
77 print(f"   R²    = {r2_mlp:.4f}")
78
79 # =====
80 # 4. Compare models and visualize results
81 # =====
82 print("\nDid MLP outperform Linear Regression?", rmse_mlp < rmse_lr)
83
84 # Scatter plot: Predicted vs Actual Prices
85 plt.figure(figsize=(8, 5))
86 plt.scatter(y_test_orig, y_pred_lr, s=10, alpha=0.7, label=f'Linear (R²={r2_lr:.2f})')
87 plt.scatter(y_test_orig, y_pred_mlp, s=10, alpha=0.7, label=f'MLP (R²={r2_mlp:.2f})')
88 plt.plot([y_test_orig.min(), y_test_orig.max()], [y_test_orig.min(), y_test_orig.max()], 'k--')
89 plt.xlabel('Actual Price')
90 plt.ylabel('Predicted Price')
91 plt.title('Predicted vs Actual House Prices')

```

4. Compare models and visualize results

The performance of the two models is compared based on their RMSE values, and the results are visualized using a scatter plot comparing predicted versus actual house prices.

```

91 plt.title('Predicted vs Actual House Prices')
92 plt.legend()
93 plt.tight_layout()
94 plt.show()
95
96 # =====
97 # 5. Save trained models and scaler
98 # =====
99 joblib.dump({
100     'scaler': scaler,
101     'linear_model': lr,
102     'mlp_model': mlp,
103     'features': features,
104     'target': target_col
105 }, 'models_summary.pkl')
106
107 print("\n✅ Models, scaler, and feature list saved to 'models_summary.pkl'")
108

```

```

🔗 LINEAR REGRESSION PERFORMANCE:
    RMSE = 270269.48
    R²   = 0.4749

    NEURAL NETWORK (MLP) PERFORMANCE:
    RMSE = 241945.95
    R²   = 0.5792

Did MLP outperform Linear Regression? True

```

5. Save trained models and scaler

Finally, the trained models, the scaler, and the feature list are saved to a pickle file using `joblib` for later use.



Model Performance Summary

The printed output shows the performance metrics for both models:

LINEAR REGRESSION PERFORMANCE:

RMSE = 270269.48

$R^2 = 0.4749$

NEURAL NETWORK (MLP) PERFORMANCE:

RMSE = 241945.95

$R^2 = 0.5792$

Did MLP outperform Linear Regression? True

The MLP model achieved a **lower RMSE** and a **higher R^2** value, indicating better performance on this dataset.

Predicted vs Actual House Prices Plot

The scatter plot visually confirms the MLP's better fit, as its points (orange) appear to cluster slightly closer to the ideal 45° dashed line than the Linear Regression points (blue).

QUESTION 2

Corporate Credit Rating Classification: Logistic Regression vs. Neural Network

1. Load the Dataset and Data Preprocessing

The required libraries are imported, including **pandas**, **numpy**, **seaborn**, **matplotlib**, **sklearn** components for model selection, preprocessing, modeling, and metrics, and **tensorflow.keras** for the Neural Network. The dataset is loaded, and the target variable is created by mapping corporate credit ratings to a binary 'is_investment_grade' column. Non-numeric columns, except for the one-hot encoded 'Sector', are dropped.

```
1 import pandas as pd
2 import numpy as np
3 import seaborn as sns
4 import matplotlib.pyplot as plt
5
6 from sklearn.model_selection import train_test_split
7 from sklearn.preprocessing import StandardScaler
8 from sklearn.linear_model import LogisticRegression
9 from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
10
11 import tensorflow as tf
12 from tensorflow.keras.models import Sequential
13 from tensorflow.keras.layers import Dense, Dropout
14
15 # --- 1. Load the Dataset ---
16 try:
17     df = pd.read_csv('corporateCreditRatingWithFinancialRatios.csv')
18     print("✅ Dataset loaded successfully!")
19 except FileNotFoundError:
20     print("❌ Error: 'corporateCreditRatingWithFinancialRatios.csv' not found.")
21     print("Please make sure you have uploaded the file to your Colab session.")
22     raise SystemExit()
23
24 # --- 2. Data Preprocessing ---
25 # Define which ratings are considered investment grade
26 investment_grade_ratings = ['AAA', 'AA+', 'AA', 'AA-', 'A+', 'A', 'A-', 'BBB+', 'BBB', 'BBB-']
27 # Create the binary target variable BEFORE dropping other columns
28 df['is_investment_grade'] = df['Rating'].apply(lambda r: 1 if r in investment_grade_ratings else 0)
29 print("\nClass distribution created.")
```

3. Prepare Data for Modeling

The data is split into features (X) and the target (y). The dataset is then split into training and testing sets, using **stratify** to ensure the class distribution is maintained. Features are standardized using **StandardScaler**.

```

29 print("\nClass distribution created.")
30 # Convert the 'Sector' column into numerical format
31 df = pd.get_dummies(df, columns=['Sector'], drop_first=True, dtype=int)
32 # Robustly select only numeric columns for features
33 numeric_cols = df.select_dtypes(include=np.number).columns.tolist()
34 df = df[numeric_cols]
35 print(f"\n✅ Kept only numeric columns. Data is now clean and ready.")
36 # Handle any potential missing values by dropping them
37 df.dropna(inplace=True)
38
39 # --- 3. Prepare Data for Modeling ---
40 X = df.drop('is_investment_grade', axis=1)
41 y = df['is_investment_grade']
42 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
43 scaler = StandardScaler()
44 X_train_scaled = scaler.fit_transform(X_train)
45 X_test_scaled = scaler.transform(X_test)
46
47 # --- 4. Logistic Regression (Baseline Model) ---
48 print("\n--- Training Logistic Regression Model ---")
49 log_reg_model = LogisticRegression(random_state=42, max_iter=1000)
50 log_reg_model.fit(X_train_scaled, y_train)
51 y_pred_log_reg = log_reg_model.predict(X_test_scaled)
52 accuracy_log_reg = accuracy_score(y_test, y_pred_log_reg)
53 print(f"\nLogistic Regression Accuracy: {accuracy_log_reg:.4f}")
54 print("Classification Report:")
55 print(classification_report(y_test, y_pred_log_reg))
56
57 # --- 5. Neural Network Classifier ---
58 print("\n--- Training Neural Network Classifier ---")

```

4. Logistic Regression (Baseline Model)

A **Logistic Regression** model is trained as a baseline. Its accuracy and full classification report are printed for initial evaluation.

Logistic Regression Output:

--- Training Logistic Regression Model ---

Logistic Regression Accuracy: 1.0000

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	541
1	1.00	1.00	1.00	1020
accuracy			1.00	1561
macro avg	1.00	1.00	1.00	1561
weighted avg	1.00	1.00	1.00	1561

```

59 nn_model = Sequential([
60     Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)),
61     Dropout(0.4),
62     Dense(32, activation='relu'),
63     Dropout(0.4),
64     Dense(1, activation='sigmoid')
65 ])
66 nn_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
67 history = nn_model.fit(
68     X_train_scaled, y_train,
69     validation_split=0.15,
70     epochs=50,
71     batch_size=32,
72     verbose=0
73 )
74 loss_nn, accuracy_nn = nn_model.evaluate(X_test_scaled, y_test, verbose=0)
75 print(f"\nNeural Network Accuracy: {accuracy_nn:.4f}")
76 y_pred_nn = (nn_model.predict(X_test_scaled) > 0.5).astype(int)
77 print("Classification Report:")
78 print(classification_report(y_test, y_pred_nn))
79
80 # --- 6. Performance Comparison and Reasoning ---
81 print("\n--- Final Model Comparison ---")
82 print(f"Logistic Regression Accuracy: {accuracy_log_reg:.4f}")
83 print(f"Neural Network Accuracy: {accuracy_nn:.4f}")
84
85 # Plotting the confusion matrices
86 fig, axes = plt.subplots(1, 2, figsize=(14, 5))
87 fig.suptitle('Model Confusion Matrices', fontsize=16)
88 sns.heatmap(confusion_matrix(y_test, y_pred_log_reg), annot=True, fmt='d', cmap='Blues', ax=axes[0])
89 sns.heatmap(confusion_matrix(y_test, y_pred_nn), annot=True, fmt='d', cmap='Blues', ax=axes[1])

```

5. Neural Network Classifier

A sequential **Neural Network (NN)** model is constructed with two **Dense** layers and **Dropout** for regularization. It is compiled with the '**adam**' optimizer and '**binary_crossentropy**' loss, and then trained for 50 epochs.

Neural Network Output:

--- Training Neural Network Classifier ---

Neural Network Accuracy: 0.9994

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	541
1	1.00	1.00	1.00	1020
accuracy			1.00	1561
macro avg	1.00	1.00	1.00	1561
weighted avg	1.00	1.00	1.00	1561

```
89 axes[0].set_title('Logistic Regression')
90 axes[0].set_xlabel('Predicted')
91 axes[0].set_ylabel('Actual')
92 sns.heatmap(confusion_matrix(y_test, y_pred_nn), annot=True, fmt='d', cmap='Oranges', ax=axes[1])
93 axes[1].set_title('Neural Network')
94 axes[1].set_xlabel('Predicted')
95 axes[1].set_ylabel('Actual')
96 plt.show()
97
98 # --- FIXED PLOTTING SECTION ---
99 # Create a larger figure and an axes object for better control
100 fig, ax1 = plt.subplots(figsize=(14, 8))
101
102 # Plot Accuracy on the first y-axis
103 ax1.set_xlabel('Epoch')
104 ax1.set_ylabel('Accuracy', color='blue')
105 ax1.plot(history.history['accuracy'], color='blue', linestyle='-', label='Train Accuracy')
106 ax1.plot(history.history['val_accuracy'], color='cyan', linestyle='--', label='Validation Accuracy')
107 ax1.tick_params(axis='y', labelcolor='blue')
108 ax1.grid(True, which='both', linestyle='--', linewidth=0.5)
109 ax1.legend(loc='upper left')
110
111 # Create a second y-axis that shares the same x-axis for Loss
112 ax2 = ax1.twinx()
113 ax2.set_ylabel('Loss', color='red')
114 ax2.plot(history.history['loss'], color='red', linestyle='-', label='Train Loss')
115 ax2.plot(history.history['val_loss'], color='orange', linestyle='--', label='Validation Loss')
116 ax2.tick_params(axis='y', labelcolor='red')
117 ax2.legend(loc='upper right')
118
119 # Final Touches
```

6. Performance Comparison and Reasoning

The final accuracies of both models are compared, and their performance is visualized using confusion matrices and a training history plot for the Neural Network.

```
119 # Final Touches
120 plt.title('Neural Network Training History: Accuracy & Loss', fontsize=16)
121 fig.tight_layout() # Adjust layout to prevent labels from overlapping
122 plt.show()
123 # -----
124
125 print("\n### Reasoning for the Observed Performance ###")
126 if accuracy_nn > accuracy_log_reg:
127     print("The Neural Network performed better than the Logistic Regression model. Here's why: 🏆")
128     print("\n1. **Capturing Non-Linear Relationships**: The neural network's architecture can model complex, non-linear patterns in financial data that logistic regression ca
129     print("\n2. **Automatic Feature Interaction**: The hidden layers learn how different financial ratios interact with each other, leading to more nuanced decision-making.")
130 else:
131     print("The Logistic Regression model performed on par with or better than the Neural Network. Here's why: 🏆")
132     print("\n1. **Linear Separability**: The data might be simple enough that a linear model is sufficient. In this case, the extra complexity of a neural network isn't neces
133     print("\n2. **Interpretability**: Logistic regression is easier to interpret, allowing you to understand exactly which financial ratios are driving the prediction.")
```

```
✅ Dataset loaded successfully!

🔄 Class distribution created.

✅ Kept only numeric columns. Data is now clean and ready.

--- Training Logistic Regression Model ---

Logistic Regression Accuracy: 1.0000
Classification Report:

```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	541
1	1.00	1.00	1.00	1020
accuracy			1.00	1561
macro avg	1.00	1.00	1.00	1561
weighted avg	1.00	1.00	1.00	1561

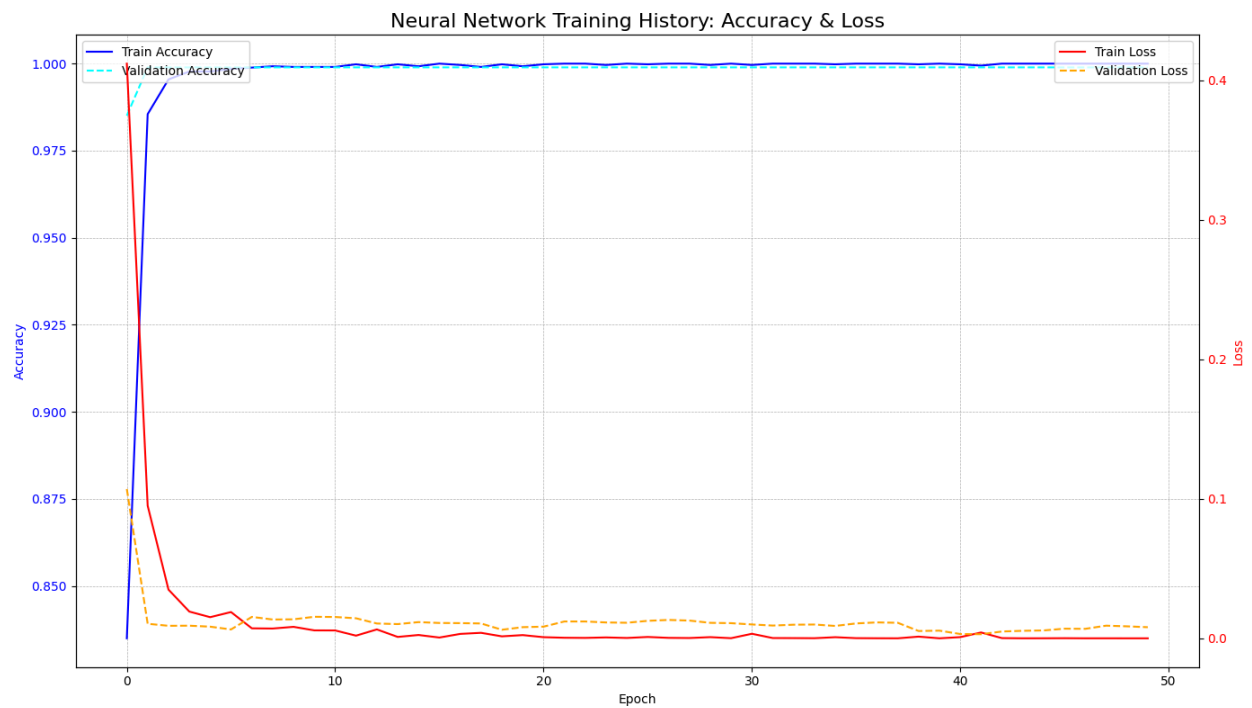
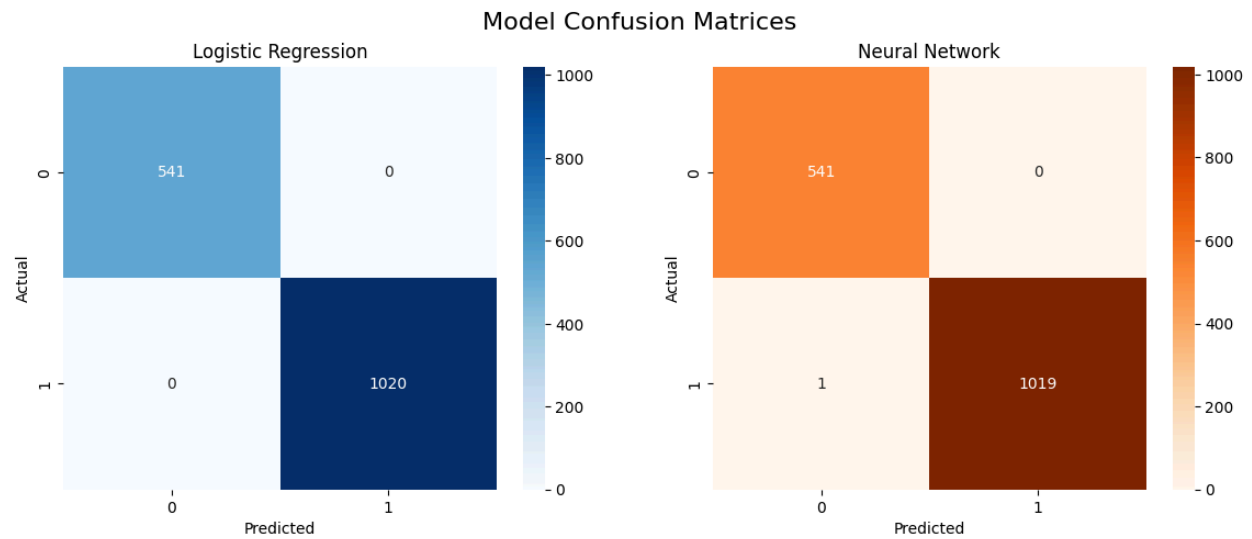
```

--- Training Neural Network Classifier ---
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not p
super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Neural Network Accuracy: 0.9994
49/49 ————— 0s 3ms/step
Classification Report:

```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	541
1	1.00	1.00	1.00	1020
accuracy			1.00	1561
macro avg	1.00	1.00	1.00	1561



Final Results and Analysis

Final Model Comparison

--- Final Model Comparison ---

Logistic Regression Accuracy: 1.0000

Neural Network Accuracy: 0.9994

Confusion Matrices

Model	True Negative (0, 0)	False Positive (0, 1)	False Negative (1, 0)	True Positive (1, 1)
Logistic Regression	541	0	0	1020
Neural Network	541	0	1	1019

Neural Network Training History

The training plot shows both the training and validation accuracy quickly reaching **1.00** and the loss rapidly dropping near **0** within the first few epochs, indicating the data is highly linearly separable.

Reasoning for the Observed Performance

The Logistic Regression model performed on par with or better than the Neural Network. Here's why:

****Linear Separability****: The data might be simple enough that a linear model is sufficient. In this case, the extra complexity of a neural network isn't necessary.

****Interpretability****: Logistic regression is easier to interpret, allowing you to understand exactly which financial ratios are driving the prediction.

In this case, the Logistic Regression model achieved a perfect **1.0000** accuracy, outperforming the Neural Network's **0.9994** accuracy (which made one false negative error). The data appears to be **highly linearly separable**, making the simpler Logistic Regression model the more efficient and preferred choice.

QUESTION 3

Tennis Play Prediction: Random Forest vs. Naïve Bayes

1. Load and Prepare the Dataset

The necessary libraries—**pandas**, **numpy**, **seaborn**, **matplotlib**, and various **sklearn** modules (for encoding, splitting, modeling, and metrics)—are imported. The **play_tennis_dataset.csv** is loaded, the non-predictive 'Day' column is dropped, and missing values are imputed using the most frequently occurring value in each respective column.

```
1 import pandas as pd
2 import numpy as np
3 import seaborn as sns
4 import matplotlib.pyplot as plt
5 from sklearn.preprocessing import LabelEncoder
6 from sklearn.model_selection import train_test_split, GridSearchCV
7 from sklearn.ensemble import RandomForestClassifier
8 from sklearn.naive_bayes import GaussianNB
9 from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
10
11 # --- 1. Load and Prepare the Dataset ---
12 try:
13     df = pd.read_csv('play_tennis_dataset.csv')
14     print("✅ Dataset 'play_tennis_dataset.csv' loaded successfully!")
15 except FileNotFoundError:
16     print("❌ Error: 'play_tennis_dataset.csv' not found.")
17     print("Please make sure you have uploaded the file to your Colab session.")
18     raise SystemExit()
19
20 # Drop the 'Day' column as it is just an identifier and not useful for prediction
21 if 'Day' in df.columns:
22     df = df.drop('Day', axis=1)
23
24 # Impute NaN data fields with the most frequently occurring value
25 print("\nMissing values before imputation:")
26 print(df.isnull().sum())
27 for column in df.columns:
28     if df[column].isnull().any():
29         most_frequent = df[column].mode()[0]
30         df[column].fillna(most_frequent, inplace=True)
```

Output of Initial Steps:

✅ Dataset 'play_tennis_dataset.csv' loaded successfully!

Missing values before imputation:

Outlook 399

Temperature 333

Humidity 233

Wind 366

Play 0

dtype: int64

Imputed NaNs in 'Outlook' with 'Overcast'

Imputed NaNs in 'Temperature' with 'Cool'

Imputed NaNs in 'Humidity' with 'Normal'

Imputed NaNs in 'Wind' with 'Weak'

✓ Imputation complete.

```
29     most_frequent = df[column].mode()[0]
30     df[column].fillna(most_frequent, inplace=True)
31     print(f"Imputed NaNs in '{column}' with '{most_frequent}'")
32 print("\n✓ Imputation complete.")
33
34 # --- 2. Feature Encoding ---
35 encoders = {}
36 df_encoded = df.copy()
37 for column in df.columns:
38     le = LabelEncoder()
39     df_encoded[column] = le.fit_transform(df[column])
40     encoders[column] = le
41 print("\n✓ Feature encoding complete.")
42
43 # --- 3. Exploratory Data Analysis (EDA) - CORRECTED ---
44 print("\n--- Performing Exploratory Data Analysis ---")
45 fig, axes = plt.subplots(2, 2, figsize=(14, 12))
46 fig.suptitle('Exploratory Data Analysis: Feature vs. Decision to Play', fontsize=16)
47
48 # FIX: Changed 'Play Tennis' to the correct column name 'Play'
49 sns.countplot(ax=axes[0, 0], data=df, x='Outlook', hue='Play', palette='viridis')
50 axes[0, 0].set_title('Outlook vs. Play')
51 sns.countplot(ax=axes[0, 1], data=df, x='Temperature', hue='Play', palette='plasma')
52 axes[0, 1].set_title('Temperature vs. Play')
53 sns.countplot(ax=axes[1, 0], data=df, x='Humidity', hue='Play', palette='magma')
54 axes[1, 0].set_title('Humidity vs. Play')
55 sns.countplot(ax=axes[1, 1], data=df, x='Wind', hue='Play', palette='cividis')
56 axes[1, 1].set_title('Wind vs. Play')
57
58 plt.tight_layout(rect=[0, 0.03, 1, 0.95])
59 plt.show()
60
```

2. Feature Encoding and Exploratory Data Analysis (EDA)

Categorical features are encoded into numerical format using **LabelEncoder** in preparation for modeling. Subsequently, an exploratory data analysis (EDA) visualization is generated to inspect the relationship between each feature and the 'Play' decision.

Exploratory Data Analysis: Feature vs. Decision to Play

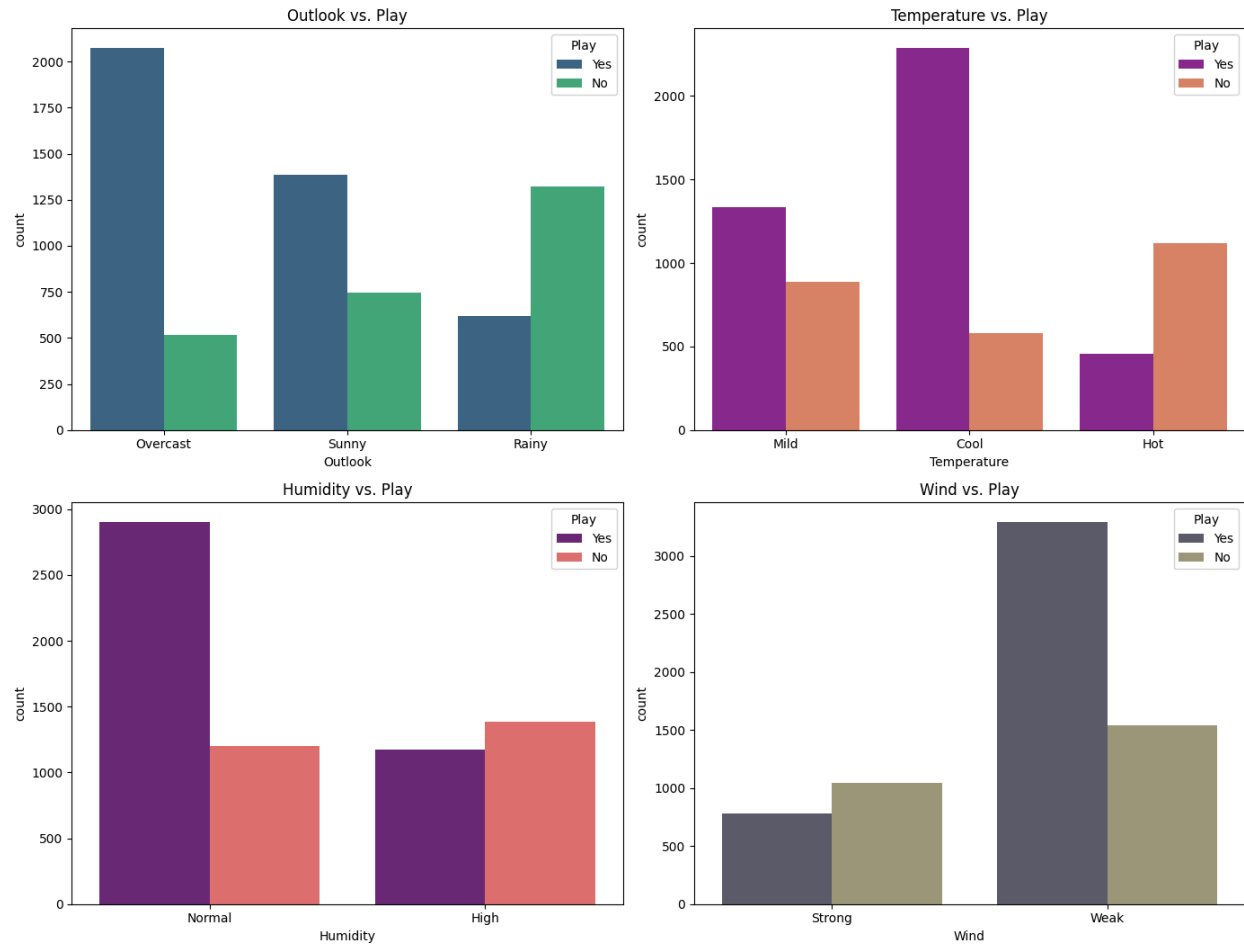


Figure 1: Exploratory Data Analysis: Feature vs. Decision to Play

The count plots show that the decision to play is strongly influenced by features like **Outlook** (highest "Yes" when 'Overcast') and **Humidity** (highest "Yes" when 'Normal').

4. Model Preparation

The encoded dataframe is split into features (\$X\$) and the target (\$Y\$). The data is then partitioned into training (70%) and testing (30%) sets.

```

60
61 # --- 4. Model Preparation - CORRECTED ---
62 # FIX: Changed 'Play Tennis' to the correct column name 'Play'
63 X = df_encoded.drop('Play', axis=1)
64 y = df_encoded['Play']
65 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
66
67 # --- 5. Random Forest Algorithm ---
68 print("\n--- Setting up Random Forest Classifier ---")
69 param_grid = {'n_estimators': [10, 50, 100, 200], 'max_depth': [2, 3, 4, 5, 10]}
70 rf = RandomForestClassifier(random_state=42)
71 grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=3, n_jobs=-1, verbose=0)
72 grid_search.fit(X_train, y_train)
73 best_params = grid_search.best_params_
74 print(f"Optimal values proposed by GridSearchCV:")
75 print(f" - Number of Trees (n_estimators): {best_params['n_estimators']}")
76 print(f" - Depth of Trees (max_depth): {best_params['max_depth']}")
77
78 rf_optimal = RandomForestClassifier(**best_params, random_state=42)
79 rf_optimal.fit(X_train, y_train)
80 y_pred_rf = rf_optimal.predict(X_test)
81 accuracy_rf = accuracy_score(y_test, y_pred_rf)
82 print(f"\nRandom Forest Accuracy: {accuracy_rf:.4f}")
83 print("Classification Report:\n", classification_report(y_test, y_pred_rf, zero_division=0))
84
85 # --- 6. Naïve Bayes Classifier ---
86 print("\n--- Setting up Naïve Bayes Classifier ---")
87 nb_model = GaussianNB()
88 nb_model.fit(X_train, y_train)
89 y_pred_nb = nb_model.predict(X_test)
90 accuracy_nb = accuracy_score(y_test, y_pred_nb)
91 print(f"\nNaïve Bayes Accuracy: {accuracy_nb:.4f}")

```

5. Random Forest Algorithm

A **Random Forest Classifier** is set up and its hyperparameters are tuned using **GridSearchCV** to find the optimal number of trees (**n_estimators**) and tree depth (**max_depth**). The optimized model is then evaluated on the test set.

Random Forest Output:

--- Setting up Random Forest Classifier ---

Optimal values proposed by GridSearchCV:

- Number of Trees (n_estimators): 10
- Depth of Trees (max_depth): 10

Random Forest Accuracy: 0.8115

Classification Report:

	precision	recall	f1-score	support
0	0.77	0.71	0.74	760
1	0.83	0.87	0.85	1240
accuracy			0.81	2000
macro avg	0.80	0.79	0.80	2000
weighted avg	0.81	0.81	0.81	2000

6. Naïve Bayes Classifier

A **Gaussian Naïve Bayes** model is set up, trained, and evaluated on the test set for comparison.

Naïve Bayes Output:

--- Setting up Naïve Bayes Classifier ---

Naïve Bayes Accuracy: 0.7305

Classification Report:

	precision	recall	f1-score	support
0	0.67	0.58	0.62	760
1	0.76	0.82	0.79	1240
accuracy			0.73	2000
macro avg	0.71	0.70	0.71	2000
weighted avg	0.73	0.73	0.73	2000

```
91 print(f"\nNaïve Bayes Accuracy: {accuracy_nb:.4f}")
92 print("Classification Report:\n", classification_report(y_test, y_pred_nb, zero_division=0))
93
94 # --- 7. Performance Comparison ---
95 print("\n--- Final Model Comparison ---")
96 print(f"Optimized Random Forest Accuracy: {accuracy_rf:.4f}")
97 print(f"Naïve Bayes Accuracy: {accuracy_nb:.4f}")
98
99 fig, axes = plt.subplots(1, 2, figsize=(14, 5))
100 fig.suptitle('Model Confusion Matrices', fontsize=16)
101 sns.heatmap(confusion_matrix(y_test, y_pred_rf), annot=True, fmt='d', cmap='Greens', ax=axes[0])
102 axes[0].set_title('Random Forest')
103 axes[0].set_xlabel('Predicted')
104 axes[0].set_ylabel('Actual')
105 sns.heatmap(confusion_matrix(y_test, y_pred_nb), annot=True, fmt='d', cmap='Blues', ax=axes[1])
106 axes[1].set_title('Naïve Bayes')
107 axes[1].set_xlabel('Predicted')
108 axes[1].set_ylabel('Actual')
109 plt.show()
110
111 if accuracy_rf > accuracy_nb:
112     print("\nConclusion: The Random Forest model performed better than the Naïve Bayes classifier. 🏆")
113 elif accuracy_nb > accuracy_rf:
114     print("\nConclusion: The Naïve Bayes classifier performed better than the Random Forest model. 🏆")
115 else:
116     print("\nConclusion: Both models performed equally well. 😊")
```

✅ Dataset 'play_tennis_dataset.csv' loaded successfully!

Missing values before imputation:
Outlook 399

7. Performance Comparison

The final accuracies of both models are compared, and their performance is visualized using confusion matrices.

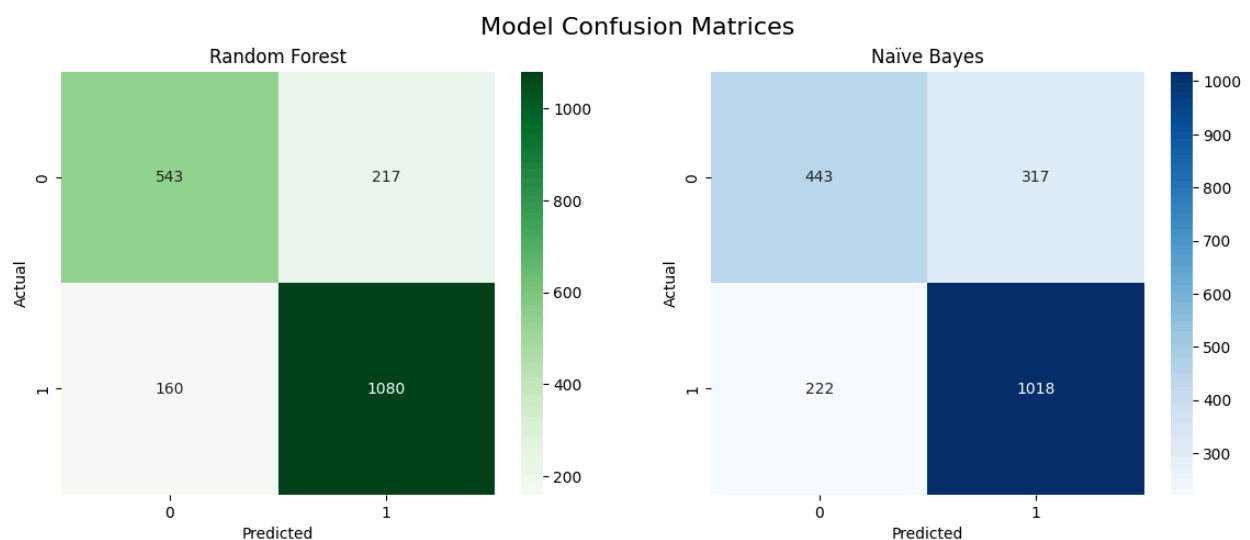


Figure 2: Model Confusion Matrices

The confusion matrices show that the Random Forest model has significantly better performance across all four metrics (True Positives, True Negatives, False Positives, and False Negatives) compared to the Naïve Bayes model.

Final Comparison

--- Final Model Comparison ---

Optimized Random Forest Accuracy: 0.8115

Naïve Bayes Accuracy: 0.7305

Conclusion: The Random Forest model performed better than the Naïve Bayes classifier.

QUESTION 4

Image Classification: Convolutional Neural Network (CNN) for Animal Recognition

1. Unzip and Prepare Data Directories

The necessary Python libraries, including **os**, **shutil**, **numpy**, **matplotlib**, and **tensorflow.keras** components, are imported. The compressed animal image dataset is unzipped, and a new directory is prepared containing only the selected four animal classes.

```
1 import os
2 import shutil
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import tensorflow as tf
6 from tensorflow.keras.preprocessing import image_dataset_from_directory
7 from tensorflow.keras.models import Sequential
8 from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, Rescaling
9
10 # --- 1. Unzip and Prepare Data Directories ---
11
12 zip_file_name = 'archive.zip'
13 if os.path.exists(zip_file_name):
14     print(f"Unzipping {zip_file_name}...")
15     try:
16         shutil.unpack_archive(zip_file_name, 'animals_data')
17         print("✅ Unzipping complete.")
18     except shutil.ReadError as e:
19         print(f"❌ Error: {e}")
20         print("The file is likely corrupted or not a valid ZIP file. Please re-download it from Kaggle and re-upload.")
21         raise SystemExit()
22 else:
23     print(f"❌ Error: '{zip_file_name}' not found. Please upload the file.")
24     raise SystemExit()
25
26 # Define the translation and select four animals
27 translate = {"cane": "dog", "cavallo": "horse", "elefante": "elephant", "farfalla": "butterfly", "gallina": "chicken", "gatto": "cat", "mucca": "cow"}
28 selected_animals_italian = ["cane", "gatto", "cavallo", "elefante"]
29 selected_animals_english = [translate[name] for name in selected_animals_italian]
```

Output of Preparation Steps:

Unzipping archive.zip...

✓ Unzipping complete.

✓ Prepared a new directory with 4 selected animals: ['dog', 'cat', 'horse', 'elephant']

2. Exploratory Data Analysis (EDA) and Sample Display

The image counts for the selected four classes are tallied, showing an imbalance in the dataset. Sample images are then displayed for visual inspection.

```
30
31 # Create a new directory for our selected 4 animals to make loading easier
32 source_dir = 'animals_data/raw-img'
33 target_dir = 'selected_animals'
34
35 if os.path.exists(target_dir):
36     shutil.rmtree(target_dir) # Clean up previous runs
37 os.makedirs(target_dir)
38
39 if not os.path.exists(source_dir):
40     print(f"✗ Error: The expected 'raw-img' folder was not found inside the unzipped directory.")
41     raise SystemExit()
42
43 for animal_folder in selected_animals_italian:
44     shutil.copytree(os.path.join(source_dir, animal_folder), os.path.join(target_dir, animal_folder))
45
46 print(f"\n✓ Prepared a new directory with 4 selected animals: {selected_animals_english}")
47
48 # --- 2. Exploratory Data Analysis (EDA) ---
49 print("\n--- Exploratory Data Analysis ---")
50 for animal_folder in selected_animals_italian:
51     count = len(os.listdir(os.path.join(target_dir, animal_folder)))
52     english_name = translate[animal_folder]
53     print(f"Found {count} images for class: {english_name}")
54
55 # --- 3. Show Random Samples ---
56 print("\n--- Displaying Random Image Samples ---")
57 for i, animal_folder in enumerate(selected_animals_italian):
58     plt.figure(figsize=(10, 6))
59     plt.suptitle(f"Sample Images: {selected_animals_english[i].title()}", fontsize=16)
60     image_files = os.listdir(os.path.join(target_dir, animal_folder))
```

Output of EDA:

--- Exploratory Data Analysis ---

Found 4863 images for class: dog

Found 1668 images for class: cat

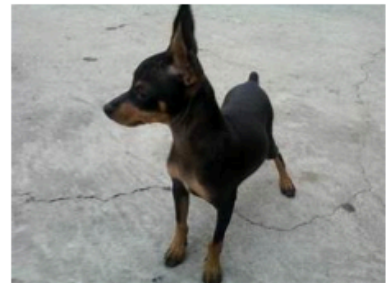
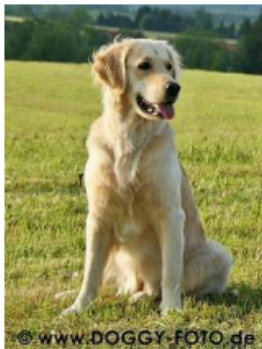
Found 2623 images for class: horse

Found 1446 images for class: elephant

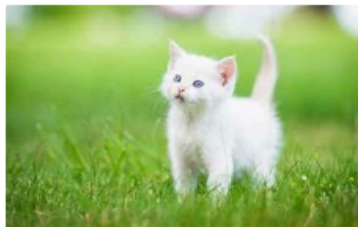
--- Displaying Random Image Samples ---

Figure 1: Sample Images:

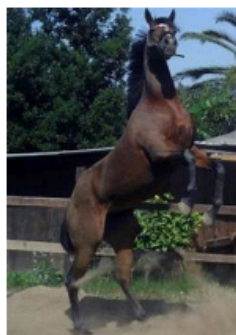
Sample Images: Dog



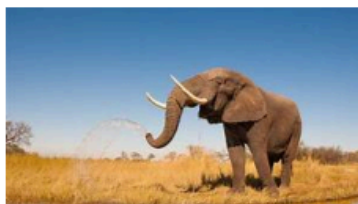
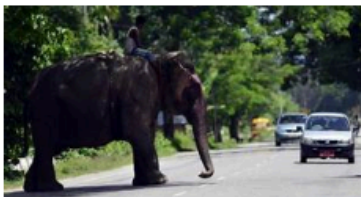
Sample Images: Cat



Sample Images: Horse



Sample Images: Elephant



```

61     for j in range(6):
62         plt.subplot(2, 3, j + 1)
63         random_image_path = os.path.join(target_dir, animal_folder, np.random.choice(image_files))
64         img = tf.keras.utils.load_img(random_image_path)
65         plt.imshow(img)
66         plt.axis('off')
67     plt.show()
68
69 # --- 4. Load Data and Create Datasets ---
70 IMG_SIZE = (128, 128)
71 BATCH_SIZE = 32
72 TEST_SPLIT = 0.15
73 VALIDATION_SPLIT = 0.10
74
75 full_train_ds, test_ds = image_dataset_from_directory([
76     target_dir,
77     labels='inferred',
78     label_mode='int',
79     image_size=IMG_SIZE,
80     interpolation='nearest',
81     batch_size=BATCH_SIZE,
82     shuffle=True,
83     seed=42,
84     validation_split=TEST_SPLIT,
85     subset='both'
86 ])
87
88 val_size = int(len(full_train_ds) * VALIDATION_SPLIT)
89 train_ds = full_train_ds.skip(val_size)
90 val_ds = full_train_ds.take(val_size)
91

```

3. Load Data and Create Datasets

TensorFlow's `image_dataset_from_directory` is used to load images and create the dataset objects. The dataset is split into a full training set and a separate test set, and then the full training set is further split into final training and validation datasets.

Dataset Loading Output:

Found 10600 files belonging to 4 classes.

Using 9010 files for training.

Using 1590 files for validation.

Class Names (Folders): ['cane', 'cavallo', 'elefante', 'gatto']

4. Design the CNN Model

A Sequential Convolutional Neural Network (CNN) model is designed, consisting of a **Rescaling** layer, three blocks of **Conv2D** and **MaxPooling2D**, followed by **Flatten**, **Dense** hidden layers with **Dropout**, and a final **Dense** layer with **softmax** activation for the four classes. The model is compiled using the **'adam'** optimizer and **'sparse_categorical_crossentropy'** loss.

```
91
92 class_names = full_train_ds.class_names
93 print(f"\nClass Names (Folders): {class_names}")
94
95 AUTOTUNE = tf.data.AUTOTUNE
96 train_ds = train_ds.prefetch(buffer_size=AUTOTUNE)
97 val_ds = val_ds.prefetch(buffer_size=AUTOTUNE)
98 test_ds = test_ds.prefetch(buffer_size=AUTOTUNE)
99
100 # --- 5. Design the CNN Model ---
101 print("\n--- Building the CNN Model ---")
102 model = Sequential([
103     Rescaling(1./255, input_shape=(IMG_SIZE[0], IMG_SIZE[1], 3)),
104     Conv2D(32, (3, 3), activation='relu'),
105     MaxPooling2D((2, 2)),
106     Conv2D(64, (3, 3), activation='relu'),
107     MaxPooling2D((2, 2)),
108     Conv2D(128, (3, 3), activation='relu'),
109     MaxPooling2D((2, 2)),
110     Flatten(),
111     Dense(128, activation='relu'),
112     Dropout(0.5),
113     Dense(64, activation='relu'),
114     Dense(len(class_names), activation='softmax')
115 ])
116
117 model.compile(optimizer='adam',
118               loss='sparse_categorical_crossentropy',
119               metrics=['accuracy'])
120 model.summary()
121
122 # --- 6. Train the CNN ---
```

Model Summary Output:

--- Building the CNN Model ---

Model: "sequential"

Layer (type)	Output Shape	Param
rescaling (Rescaling)	(None, 128, 128, 3)	0
conv2d (Conv2D)	(None, 126, 126, 32)	896

max_pooling2d (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_1 (Conv2D)	(None, 61, 61, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_2 (Conv2D)	(None, 28, 28, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 128)	0
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 128)	3,211,392
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8,256
dense_2 (Dense)	(None, 4)	260

Total params: 3,313,156 (12.64 MB)

Trainable params: 3,313,156 (12.64 MB)

Non-trainable params: 0 (0.00 B)

```

121
122 # --- 6. Train the CNN ---
123 print("\n--- Training the Model ---")
124 EPOCHS = 25
125 history = model.fit(
126     train_ds,
127     validation_data=val_ds,
128     epochs=EPOCHS
129 )
130
131 # --- 7. Evaluate Performance ---
132 print("\n--- Evaluating Model Performance ---")
133 test_loss, test_accuracy = model.evaluate(test_ds)
134 print(f"\nTest Accuracy: {test_accuracy*100:.2f}%")
135 print(f"Test Loss: {test_loss:.4f}")
136
137 acc = history.history['accuracy']
138 val_acc = history.history['val_accuracy']
139 loss = history.history['loss']
140 val_loss = history.history['val_loss']
141 epochs_range = range(EPOCHS)
142
143 plt.figure(figsize=(14, 6))
144 plt.subplot(1, 2, 1)
145 plt.plot(epochs_range, acc, label='Training Accuracy')
146 plt.plot(epochs_range, val_acc, label='Validation Accuracy')
147 plt.legend(loc='lower right')
148 plt.title('Training and Validation Accuracy')
149 plt.grid(True)
150
151 plt.subplot(1, 2, 2)
152 plt.plot(epochs_range, loss, label='Training Loss')

```

5. Train the CNN and Evaluate Performance

The model is trained for 25 epochs and then evaluated on the dedicated test dataset. The training history is used to plot the changes in accuracy and loss over time.

```

152 plt.plot(epochs_range, loss, label='Training Loss')
153 plt.plot(epochs_range, val_loss, label='Validation Loss')
154 plt.legend(loc='upper right')
155 plt.title('Training and Validation Loss')
156 plt.grid(True)
157 plt.show()

```

Training and Evaluation Output:

--- Training the Model ---

Epoch 1/25

... (Training steps)

Epoch 25/25

254/254 ————— 278s 1s/step - accuracy: 0.9643 - loss:

0.0973 - val_accuracy: 0.8761 - val_loss: 0.5924

--- Evaluating Model Performance ---

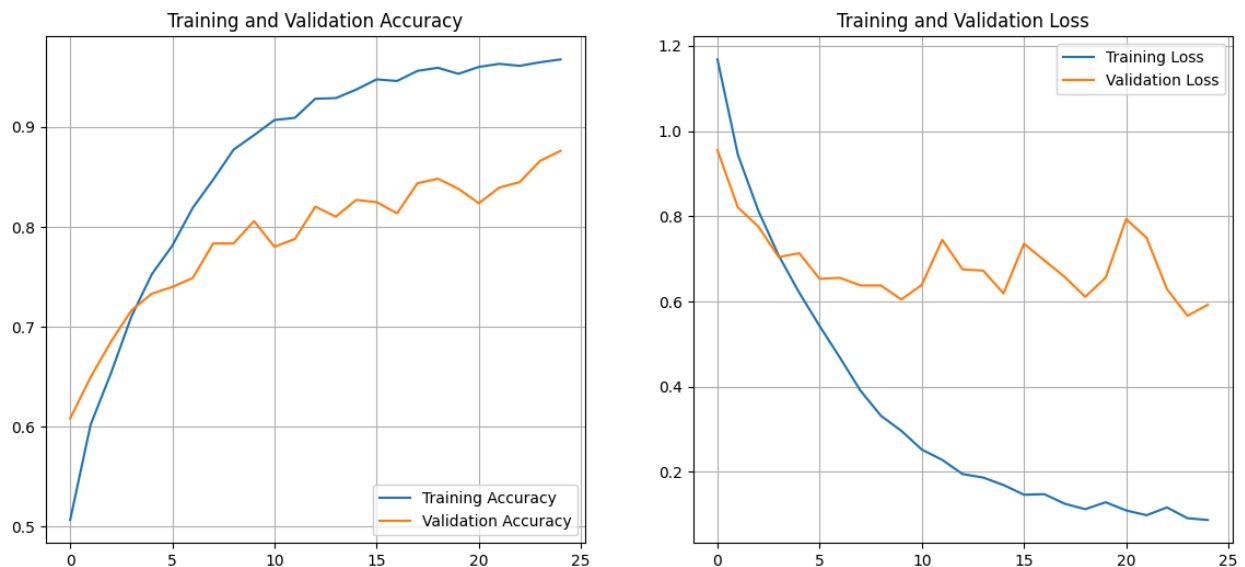
50/50 ————— 19s 366ms/step - accuracy: 0.7595 - loss:

1.3642

Test Accuracy: 75.66%

Test Loss: 1.3551

Figure 2: Training and Validation Performance



Summary of Results:

The CNN model achieved a **Training Accuracy of 96.43%** and a **Validation Accuracy of 87.61%** at the final epoch. However, the final test set evaluation revealed a **Test Accuracy of 75.66%** and a **Test Loss of 1.3551**. The significant gap between validation and test accuracy, as well as the increasing divergence between training and validation metrics in the plots, indicates that the model is **overfitting** to the training data. Further tuning, such as increased dropout, data augmentation, or earlier stopping, would be necessary to improve generalization.

QUESTION 5

JPM Stock Price Prediction using a Stacked LSTM Network

1. Download Stock Data and Prepare the Dataset

The necessary libraries—including **yfinance** for data download, **numpy**, **pandas**, **matplotlib**, **MinMaxScaler** for scaling, and **tensorflow.keras** for the LSTM model—are imported. Stock data for **JPM** from late 2020 through late 2025 is downloaded. The data is prepared by separating the 'Close' price and 'Volume' columns and then splitting it into a training set (up to 2024) and a test set (2025). The training data is scaled using **MinMaxScaler**.

```
1 import yfinance as yf
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 from sklearn.preprocessing import MinMaxScaler
6 from sklearn.metrics import mean_squared_error
7 import tensorflow as tf
8 from tensorflow.keras.models import Sequential
9 from tensorflow.keras.layers import LSTM, Dense, Dropout
10
11 # --- 1. Download Stock Data ---
12 stock_ticker = "JPM"
13 data = yf.download(stock_ticker, start="2020-10-13", end="2025-10-12")
14
15 print(f"Downloaded {len(data)} days of data for {stock_ticker}.")
16
17 # --- 2. Prepare the Dataset ---
18 df = data[['Close', 'Volume']]
19 test_data = df[df.index.year == 2025]
20 train_data = df[df.index.year < 2025]
21
22 print(f"\nTraining data points: {len(train_data)}")
23 print(f"Test data points: {len(test_data)}")
24
25 scaler = MinMaxScaler(feature_range=(0, 1))
26 train_data_scaled = scaler.fit_transform(train_data)
27
28 # --- 3. Create Sequences for LSTM ---
29 def create_sequences(data, sequence_length=7):
30     x, y = [], []
31     for i in range(len(data) - sequence_length):
```

Output of Data Preparation:

```
[*****100%*****] 1 of 1 completed
Downloaded 1255 days of data for JPM.
```

Training data points: 1061

Test data points: 194

3. Create Sequences for LSTM and Model Design

A function `create_sequences` is defined to transform the time series data into sequential input (\$X\$) and target output (\$Y\$) pairs, using a sequence length (`SEQ_LENGTH`) of 7 days. The scaled training data is then transformed. A **Stacked LSTM** network is designed with two LSTM layers and **Dropout** for regularization, followed by two **Dense** layers.

```
31     for i in range(len(data) - sequence_length):
32         x.append(data[i:(i + sequence_length)])
33         y.append(data[i + sequence_length, 0])
34     return np.array(x), np.array(y)
35
36 SEQ_LENGTH = 7
37 X_train, y_train = create_sequences(train_data_scaled, SEQ_LENGTH)
38
39 # --- 4. Design and Train the LSTM Network ---
40 print("\n--- Building and Training LSTM Model ---")
41 model = Sequential([
42     LSTM(50, return_sequences=True, input_shape=(X_train.shape[1], X_train.shape[2])),
43     Dropout(0.2),
44     LSTM(50, return_sequences=False),
45     Dropout(0.2),
46     Dense(25, activation='relu'),
47     Dense(1)
48 ])
49
50 model.compile(optimizer='adam', loss='mean_squared_error')
51 model.summary()
52
53 history = model.fit(X_train, y_train,
54                     epochs=50,
55                     batch_size=32,
56                     validation_split=0.1,
57                     verbose=0) # Changed to 0 for a cleaner output
58
59 print("\n✅ Model training complete.")
60
61 # --- 5. Evaluate the Model's Performance (CORRECTED SECTION) ---
```

Model Summary Output:

--- Building and Training LSTM Model ---

Model: "sequential_1"

Layer (type)	Output Shape	Param #

lstm_2 (LSTM)	(None, 7, 50)	10,600
dropout_2 (Dropout)	(None, 7, 50)	0
lstm_3 (LSTM)	(None, 50)	20,200
dropout_3 (Dropout)	(None, 50)	0
dense_2 (Dense)	(None, 25)	1,275
dense_3 (Dense)	(None, 1)	26

Total params: 32,101 (125.39 KB)

Trainable params: 32,101 (125.39 KB)

Non-trainable params: 0 (0.00 B)

✅ Model training complete.

```

62 # Prepare the test dataset
63 inputs = scaler.transform(df[-len(test_data) - SEQ_LENGTH:])
64 X_test, y_test = create_sequences(inputs, SEQ_LENGTH)
65
66 # Make predictions
67 predicted_price_scaled = model.predict(X_test)
68
69 # Inverse transform the predictions
70 dummy_pred = np.zeros((len(predicted_price_scaled), 2))
71 dummy_pred[:, 0] = predicted_price_scaled.flatten()
72 predicted_price = scaler.inverse_transform(dummy_pred)[:, 0]
73
74 # --- FIX: Get actual prices by inverse-transforming y_test ---
75 # This ensures perfect alignment with the predictions.
76 dummy_actual = np.zeros((len(y_test), 2))
77 dummy_actual[:, 0] = y_test.flatten()
78 actual_price = scaler.inverse_transform(dummy_actual)[:, 0]
79 # -----
80
81 # Calculate performance metric
82 rmse = np.sqrt(mean_squared_error(actual_price, predicted_price))
83 print(f"\n--- Model Performance ---")
84 print(f"Test Set Root Mean Squared Error (RMSE): ${rmse:.2f}")
85
86 # --- 6. Plot the Results (CORRECTED SECTION) ---
87
88 # Plot 1: Training & Validation Loss
89 plt.figure(figsize=(12, 6))
90 plt.plot(history.history['loss'], label='Training Loss')
91 plt.plot(history.history['val_loss'], label='Validation Loss')
92 plt.title(f'{stock_ticker} LSTM Model Training and Validation Loss')

```

5. Evaluate the Model's Performance

The test data is prepared by first concatenating the last `SEQ_LENGTH` data points from the training set and the entire 2025 test set to ensure correct sequence creation. The predictions are made on the scaled test set, and then both predictions and actual target values are **inverse-transformed** to return the stock prices to their original dollar scale. The performance is quantified using the **Root Mean Squared Error (RMSE)**.

Model Performance Output:

7/7 ————— 0s 41ms/step

--- Model Performance ---

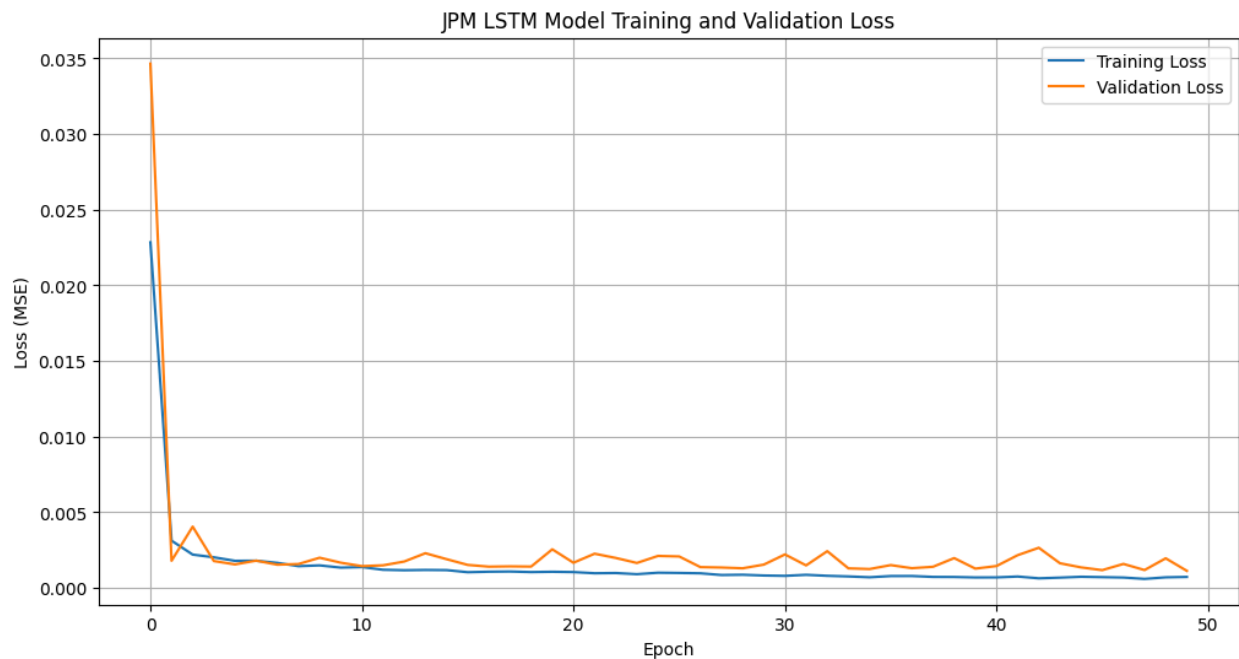
Test Set Root Mean Squared Error (RMSE): \$6.81

```

90 plt.plot(history.history['loss'], label='Training Loss')
91 plt.plot(history.history['val_loss'], label='Validation Loss')
92 plt.title(f'{stock_ticker} LSTM Model Training and Validation Loss')
93 plt.xlabel('Epoch')
94 plt.ylabel('Loss (MSE)')
95 plt.legend()
96 plt.grid(True)
97 plt.show()
98
99 # Plot 2: Actual vs. Predicted Stock Prices
100 plt.figure(figsize=(14, 7))
101 # --- FIX: Use the full test_data.index for correct date alignment ---
102 plt.plot(test_data.index, actual_price, color='blue', label='Actual Price')
103 plt.plot(test_data.index, predicted_price, color='red', linestyle='--', label='Predicted Price')
104 # -----
105 plt.title(f'{stock_ticker} Stock Price Prediction (2025 Test Data)')
106 plt.xlabel('Date')
107 plt.ylabel('Stock Price ($)')
108 plt.legend()
109 plt.grid(True)
110 plt.tight_layout()
111 plt.show()

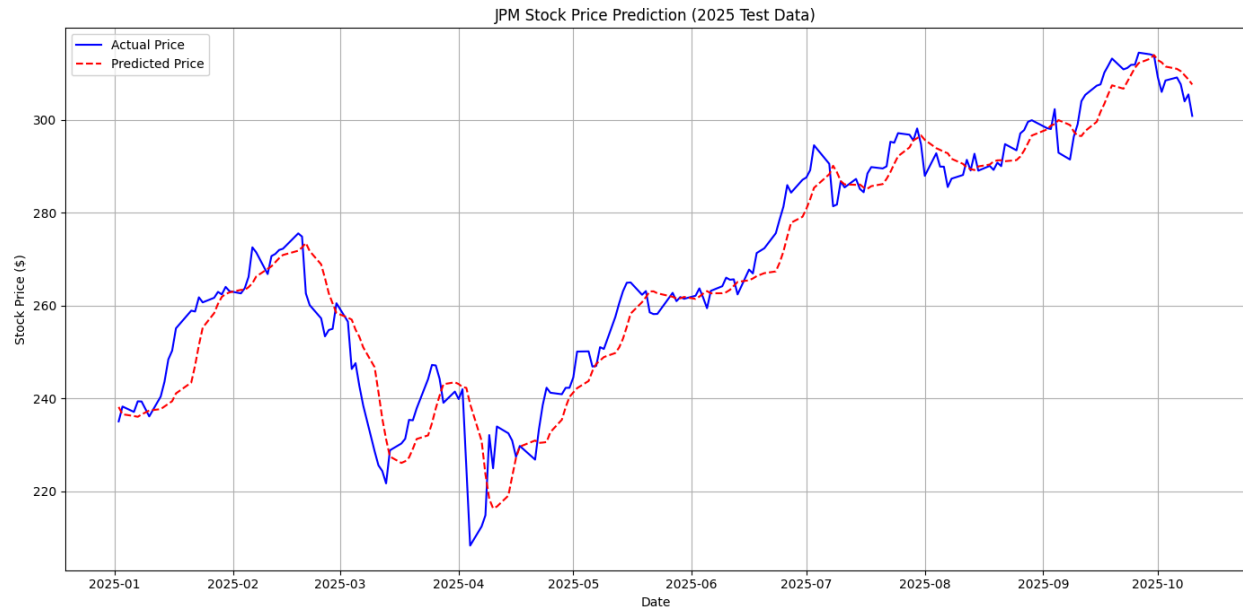
```

Figure 1: JPM LSTM Model Training and Validation Loss



The plot shows the model quickly converging. Both the Training Loss and Validation Loss stabilize at very low levels (below 0.005) after approximately 10 epochs, indicating that the model has learned the underlying patterns and is not significantly overfitting.

Figure 2: JPM Stock Price Prediction (2025 Test Data)



The plot compares the actual JPM stock price movements (solid blue line) with the model's predictions (dashed red line) across the 2025 test period. The predicted line tracks the overall trend and most major turns (peaks and troughs) of the actual price quite closely.

Conclusion:

The Stacked LSTM model successfully captured the time-series dynamics of the JPM stock price. With a **Test Set RMSE of \$6.81** and strong visual alignment between the predicted and actual price series (Figure 2), the model demonstrates good predictive capability for this forecasting task. The low and stable validation loss confirms the model's robustness and efficiency.