BU425 Final Project

Created by:
Amy Freij Camacho – 211912090
Wang Qianyi – 169036969
Nicholas Rebello – 210572430
Yash Harshal Barve - 211544920

Our project comprises two parts:

1. Resource Allocation and Yield Prediction
2. Yield Prediction and Risk Mitigation

Part 1 focuses on extracting the most important features that affect the yield. This is done using a Random Forest Regressor. Comprehensive visualizations are included to study the dependence of the yield on every feature variable. Additionally, we used Lasso Regression to predict the yield, to ensure that the selected features actually improve the accuracy of the predictions.

Part 2 focuses on actually predicting the yield and calculating a risk score based on the feature variables and the predicted yield. This is a temporal dataset (time-series). Therefore, an LSTM model would work well here. To take advantage of the temporal property of the dataset, and to reduce the noise, we added a few derived variables using common feature engineering techniques. We then used a Random Forest Regressor to extract the most important features (just like part 1). Next, we trained an LSTM model using these features. We analyzed the outputs and developed a customized function to calculate the risk score based on the model predictions as well as historical data.

Visualizing the dataset

In [3]:

```
# import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# importing the dataset

# dataset link: https://www.kaggle.com/datasets/patelris/crop-yield-predi

yield_df = pd.read_csv("yield_df.csv")
print(yield_df.shape)
```

```
(28242, 8)
```

In [5]:

```
# Display basic summary
print("Basic Info and Summary Statistics:")
```

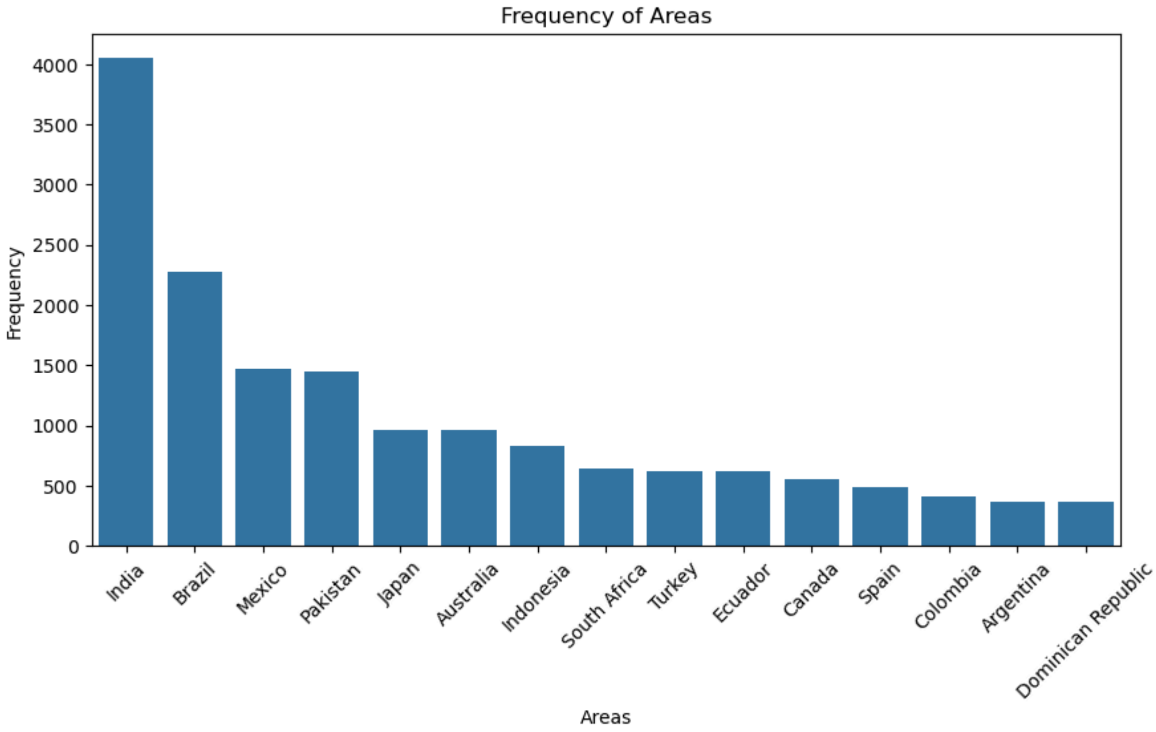
```
print(yield_df.describe(include='all'))
```

```
Basic Info and Summary Statistics:
          Unnamed: 0   Area      Item          Year     hg/ha_yield  \
count   28242.000000  28242     28242  28242.000000   28242.000000
unique           NaN    101        10           NaN            NaN
top              NaN  India  Potatoes           NaN            NaN
freq             NaN   4048      4276           NaN            NaN
mean    14120.500000    NaN       NaN   2001.544296   77053.332094
std      8152.907488    NaN       NaN      7.051905   84956.612897
min         0.000000    NaN       NaN   1990.000000      50.000000
25%      7060.250000    NaN       NaN   1995.000000   19919.250000
50%     14120.500000    NaN       NaN   2001.000000   38295.000000
75%     21180.750000    NaN       NaN   2008.000000  104676.750000
max     28241.000000    NaN       NaN   2013.000000  501412.000000

        average_rain_fall_mm_per_year  pesticides_tonnes      avg_temp
count                     28242.00000       28242.000000  28242.000000
unique                            NaN                NaN           NaN
top                               NaN                NaN           NaN
freq                              NaN                NaN           NaN
mean                       1149.05598       37076.909344     20.542627
std                         709.81215       59958.784665      6.312051
min                          51.00000           0.040000      1.300000
25%                         593.00000        1702.000000     16.702500
50%                        1083.00000       17529.440000     21.510000
75%                        1668.00000       48687.880000     26.000000
max                        3240.00000      367778.000000     30.650000
```

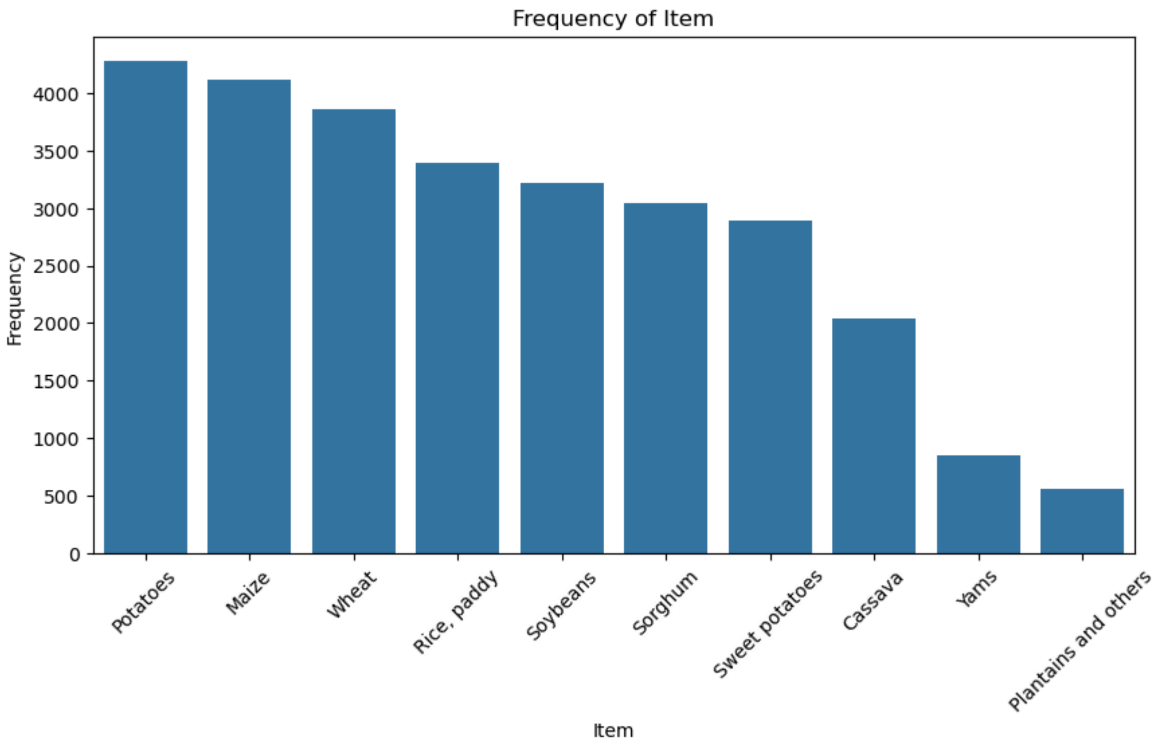
In [7]:
```
# proportion of missing data
print(yield_df.isnull().sum() / len(yield_df))
```

```
Unnamed: 0                       0.0
Area                             0.0
Item                             0.0
Year                             0.0
hg/ha_yield                      0.0
average_rain_fall_mm_per_year    0.0
pesticides_tonnes                0.0
avg_temp                         0.0
dtype: float64
```

In [9]:
```
# frequency histogram of areas
plt.figure(figsize=(10, 5))
sns.countplot(data=yield_df, x='Area', order=yield_df['Area'].value_count
plt.title('Frequency of Areas')
plt.xlabel('Areas')
plt.ylabel('Frequency')
plt.xticks(rotation=45)
plt.show()
```

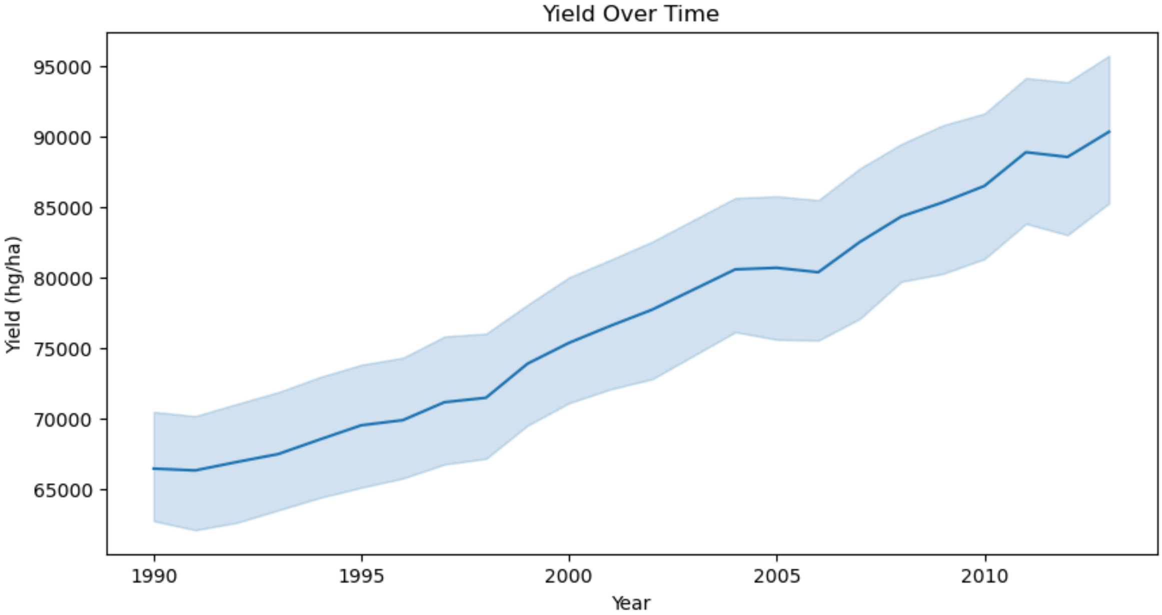


In [11]:
```
# frequency histogram of items
plt.figure(figsize=(10, 5))
sns.countplot(data=yield_df, x='Item', order=yield_df['Item'].value_count
plt.title('Frequency of Item')
plt.xlabel('Item')
plt.ylabel('Frequency')
plt.xticks(rotation=45)
plt.show()
```

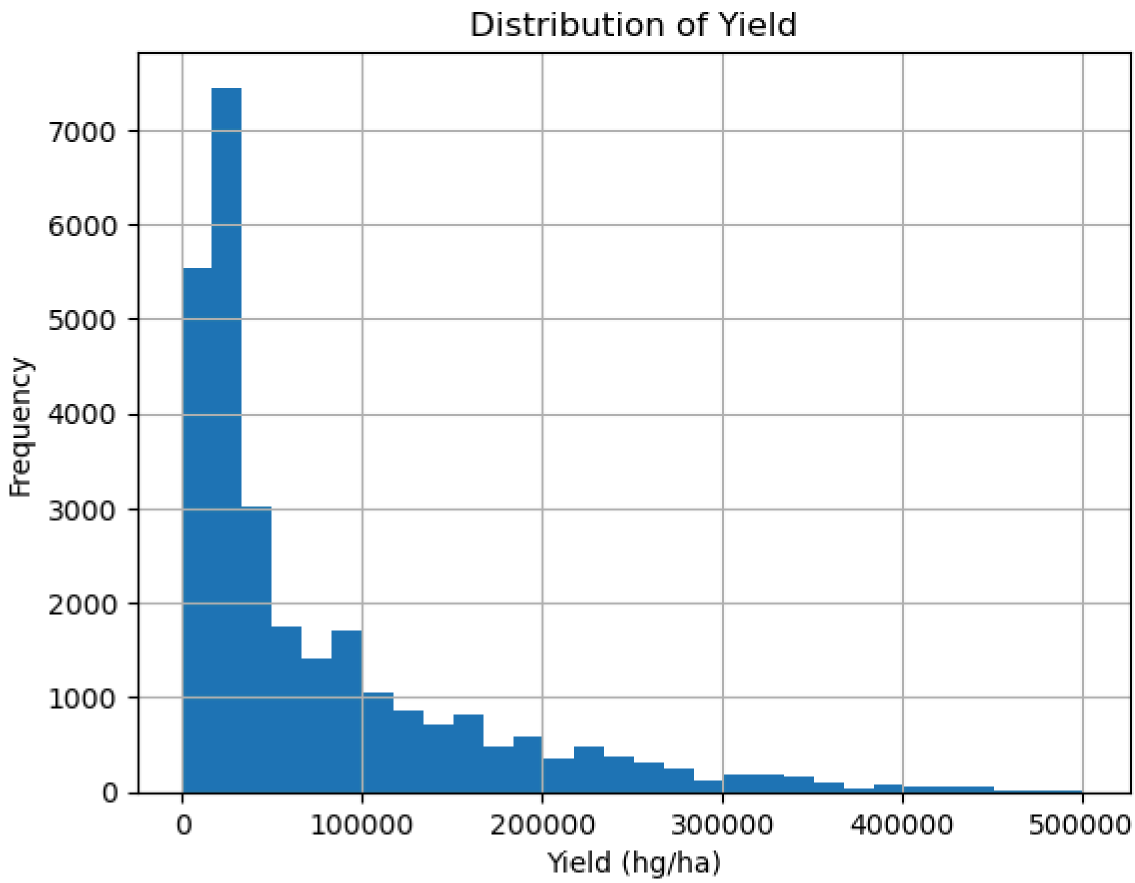


In [13]:
```
# yield over time
plt.figure(figsize=(10, 5))
sns.lineplot(data=yield_df, x='Year', y='hg/ha_yield')
plt.title('Yield Over Time')
plt.xlabel('Year')
```

```
plt.ylabel('Yield (hg/ha)')
plt.show()
```

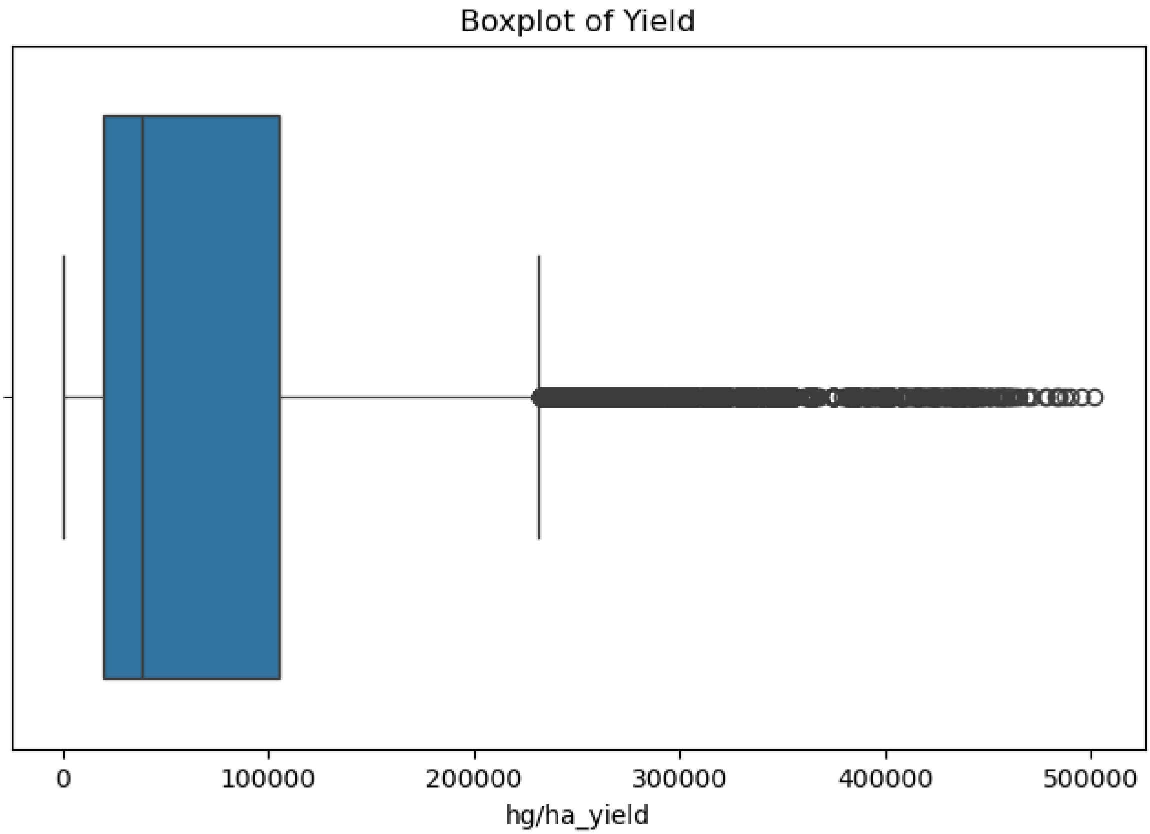


In [15]:
```
# distribution of yield
yield_df['hg/ha_yield'].hist(bins=30)
plt.title('Distribution of Yield')
plt.xlabel('Yield (hg/ha)')
plt.ylabel('Frequency')
plt.show()
```



In [17]:
```
# box plot of yield
plt.figure(figsize=(8, 5))
sns.boxplot(x=yield_df['hg/ha_yield'])
```

```
plt.title('Boxplot of Yield')
plt.show()
```



Part I: Resource Allocation and Yield Prediction

In [19]:

```
# importing libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Lasso
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_s
import matplotlib.pyplot as plt
import seaborn as sns

# dataset link: https://www.kaggle.com/datasets/patelris/crop-yield-predi

original_resource_cols = ["average_rain_fall_mm_per_year", "pesticides_to
X_original = yield_df[original_resource_cols].copy()
```

Data Preprocessing

In [21]:

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

# Encode categorical variables (Area and Item)
yield_df = pd.get_dummies(yield_df, columns=["Area", "Item"], drop_first=

# Normalize numerical features
```

```
scaler = StandardScaler()
numerical_cols = ["average_rain_fall_mm_per_year", "pesticides_tonnes", "
yield_df[numerical_cols] = scaler.fit_transform(yield_df[numerical_cols])
```

```
Unnamed: 0                       0
Area                             0
Item                             0
Year                             0
hg/ha_yield                      0
average_rain_fall_mm_per_year    0
pesticides_tonnes                0
avg_temp                         0
dtype: int64
```

Feature Engineering - Adding derived features

In [23]:

```
# Create interaction features
yield_df["rainfall_temp_interaction"] = yield_df["average_rain_fall_mm_pe

# Create crop-specific features (avoid division by zero)
yield_df["yield_per_rainfall"] = yield_df["hg/ha_yield"] / (yield_df["ave
yield_df["yield_per_pesticide"] = yield_df["hg/ha_yield"] / (yield_df["pe
```

Split the Data into Training and Testing Sets

In [25]:

```
X = yield_df.drop(columns=["hg/ha_yield"])  # Features
y = yield_df["hg/ha_yield"]  # Target variable

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
```

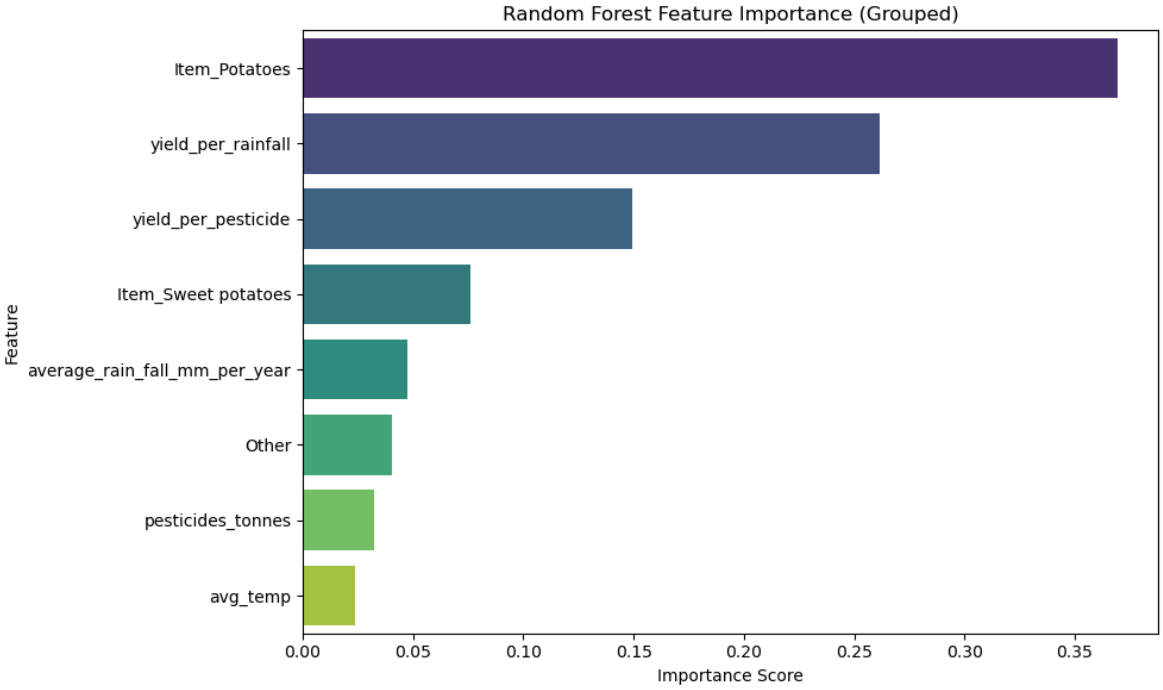
Feature Selection (Based on Random Forest Feature Importance)

In [27]:

```
# Feature Selection using Random Forest
rf_model = RandomForestRegressor(random_state=42)
rf_model.fit(X_train, y_train)
importances = rf_model.feature_importances_
feature_names = X.columns

# Create feature importance DataFrame
feature_importance_df = pd.DataFrame({"Feature": feature_names, "Importan
feature_importance_df = feature_importance_df.sort_values(by="Importance"

# Group low-importance features
threshold = 0.01
important_features_df = feature_importance_df[feature_importance_df["Impo
low_importance_df = feature_importance_df[feature_importance_df["Importan
other_importance = low_importance_df["Importance"].sum()
other_row = pd.DataFrame([{"Feature": "Other", "Importance": other_import
final_feature_df = pd.concat([important_features_df, other_row], ignore_i
final_feature_df = final_feature_df.sort_values(by="Importance", ascendin

print("\nFinal Feature Importance DataFrame:")
print(final_feature_df.to_string(index=False))

# Plot feature importance
plt.figure(figsize=(10, 6))
```

```
sns.barplot(x="Importance", y="Feature", data=final_feature_df, hue="Feat
plt.title("Random Forest Feature Importance (Grouped)")
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.tight_layout()
plt.show()
```

```
Final Feature Importance DataFrame:
                         Feature  Importance
                   Item_Potatoes    0.369278
              yield_per_rainfall    0.261532
             yield_per_pesticide    0.149455
             Item_Sweet potatoes    0.076166
   average_rain_fall_mm_per_year    0.047264
                           Other    0.040533
               pesticides_tonnes    0.032164
                        avg_temp    0.023610
```



Keep Only Important Features for Model Training

In [29]:

```
important_features = important_features_df["Feature"]
X_train_important = X_train[important_features]
X_test_important = X_test[important_features]

print("Features used in the model (with importance scores):")
for _, row in important_features_df.iterrows():
    print(f"- {row['Feature']}: {row['Importance']:.4f}")
```

```
Features used in the model (with importance scores):
- Item_Potatoes: 0.3693
- yield_per_rainfall: 0.2615
- yield_per_pesticide: 0.1495
- Item_Sweet potatoes: 0.0762
- average_rain_fall_mm_per_year: 0.0473
- pesticides_tonnes: 0.0322
- avg_temp: 0.0236
```

Train the Lasso Model with Selected Features

In [31]:
```
# GridSearchCV for optimal Lasso alpha
lasso = Lasso(random_state=42, max_iter=10000)
param_grid = {"alpha": [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]}
grid = GridSearchCV(lasso, param_grid, cv=5, scoring="neg_mean_squared_er
grid.fit(X_train_important, y_train)

best_lasso = grid.best_estimator_
print("Best Lasso alpha:", grid.best_params_["alpha"])

# Predict and evaluate
y_pred = best_lasso.predict(X_test_important)

mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

print("\nModel Evaluation with Selected Features (Lasso):")
print(f"MAE:  {mae:.2f}")
print(f"MSE:  {mse:.2f}")
print(f"RMSE: {rmse:.2f}")
print(f"R²:   {r2:.4f}")

# Print Lasso Coefficients
coef_df = pd.DataFrame({
    "Feature": X_train_important.columns,
    "Coefficient": best_lasso.coef_
}).sort_values(by="Coefficient", key=abs, ascending=False)

print("\nYield Sensitivity (Lasso Coefficients):")
print(coef_df.to_string(index=False))

# Positive coef → ↑ resource → ↑ yield
# Negative coef → ↑ resource → ↓ yield
# Bigger magnitude → more sensitive
```

```
Best Lasso alpha: 0.001

Model Evaluation with Selected Features (Lasso):
MAE:  41379.71
MSE:  3689170612.38
RMSE: 60738.54
R²:   0.4914

Yield Sensitivity (Lasso Coefficients):
                      Feature   Coefficient
                 Item_Potatoes 149464.776756
           Item_Sweet potatoes  72144.766843
                      avg_temp  -9809.565921
             pesticides_tonnes   5996.628954
average_rain_fall_mm_per_year   5893.771020
             yield_per_rainfall     -0.019336
            yield_per_pesticide      0.000453
```

Interpretation of Model Performance

The Lasso regression model identified key drivers of crop yield: **potatoes** and **sweet potatoes** had the strongest positive impact, highlighting the importance of crop selection. **Rainfall and pesticide use positively influenced yield, while higher temperatures reduced productivity, suggesting potential heat stress effects. These insights support optimized resource allocation and climate-resilient farming strategies.**

After tuning via **cross-validation**, the model was evaluated on a hold-out test set using standard regression metrics:

MAE: 41379.71 hg/ha

On average, the model's predictions deviate from actual crop yield by around 41,000 hg/ha. This gives a tangible estimate of prediction error.

MSE: 3689170612.38 hg/ha

Squared error penalizes larger mistakes more. While less interpretable directly, it helps in optimizing the model during training.

RMSE: 60738.54 hg/ha

It shows the typical size of prediction error. RMSE is higher than MAE because it accounts for larger errors more heavily.

The error margins are acceptable for high-variance agricultural data.

R^2 Score: 0.4914

The model explains about 49.1% of the variance in crop yield. While not perfect, this is considered reasonable for real-world agricultural data, where external factors like soil quality, disease, and farming practices add noise.

Overall, the model effectively captures the main yield drivers and provides reliable, interpretable insights for guiding resource allocation decisions.

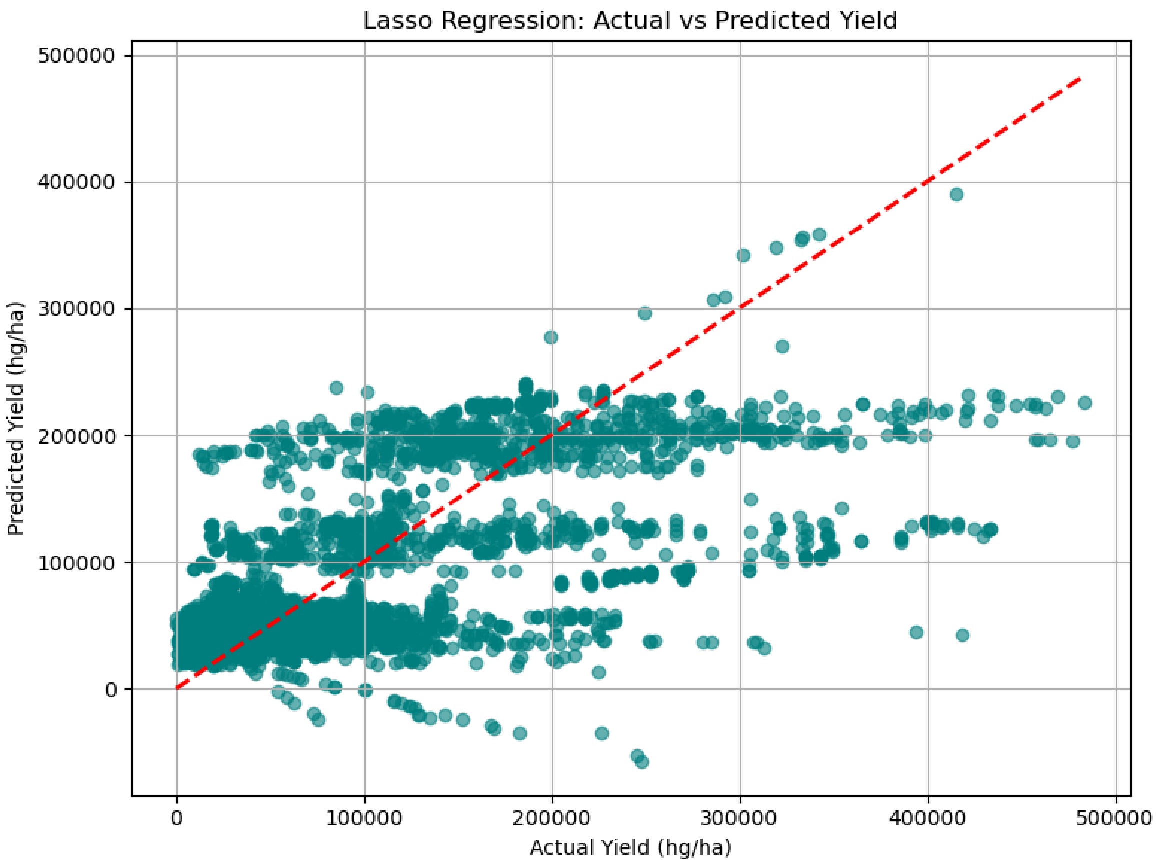
The Lasso regression model revealed several important insights into the drivers of crop yield. **Crop type, particularly potatoes and sweet potatoes, showed the strongest positive influence on yield, emphasizing the importance of selecting high performing crops for resource allocation. Among environmental and resource related variables, rainfall and pesticide use had a positive impact on predicted yield, while average temperature had a negative effect, indicating that higher temperatures may reduce agricultural productivity.**

The model also highlighted the role of resource efficiency, with features like **yield_per_rainfall** and **yield_per_pesticide** contributing to prediction, though with smaller coefficients. This suggests that not only the quantity, but also the effectiveness of input usage, matters when optimizing yield.

Overall, the model provides a useful, interpretable framework for understanding how different agricultural inputs influence crop yield, supporting more informed and data-driven decisions around resource allocation, climate impact mitigation, and crop selection.

Visualizing the model's predictions

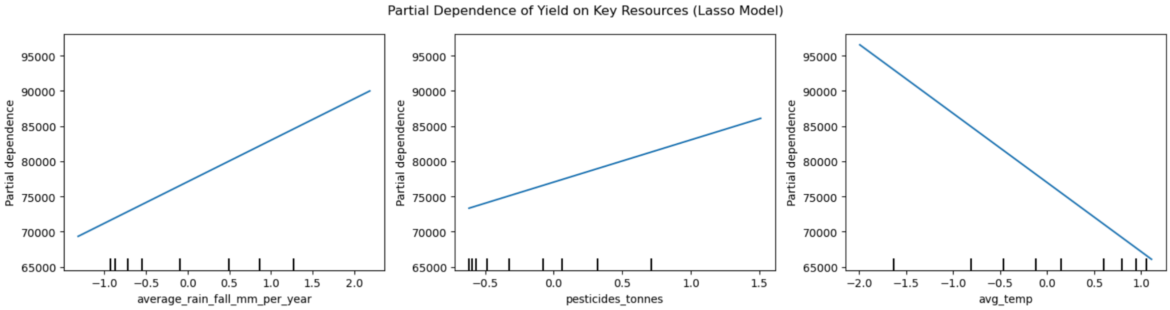
In [33]:
```
# Actual vs Predicted Yield
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, alpha=0.6, color='teal')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--
plt.xlabel("Actual Yield (hg/ha)")
plt.ylabel("Predicted Yield (hg/ha)")
plt.title("Lasso Regression: Actual vs Predicted Yield")
plt.tight_layout()
plt.grid(True)
plt.show()
```



The scatter plot of actual vs. predicted yields from the Lasso Regression model shows a moderate fit, with an R^2 of 0.4914, indicating that the model explains about 49% of the variance in yields. The spread of points around the line of perfect prediction suggests variability in model accuracy, particularly at higher yield levels. This variability and apparent patterns of under and overprediction may benefit from further model refinement to enhance prediction accuracy and consistency across the range of yields.

Marginal Effect of Key Resources on Predicted Yield

In [35]:
```
from sklearn.inspection import PartialDependenceDisplay
fig, ax = plt.subplots(1, 3, figsize=(15, 4))

features_to_plot = ["average_rain_fall_mm_per_year", "pesticides_tonnes",
PartialDependenceDisplay.from_estimator(best_lasso, X_train_important, fe
plt.suptitle("Partial Dependence of Yield on Key Resources (Lasso Model)"
plt.tight_layout()
plt.show()
```



These partial dependence plots illustrate how key resources impact crop yield according to a Lasso Regression model:

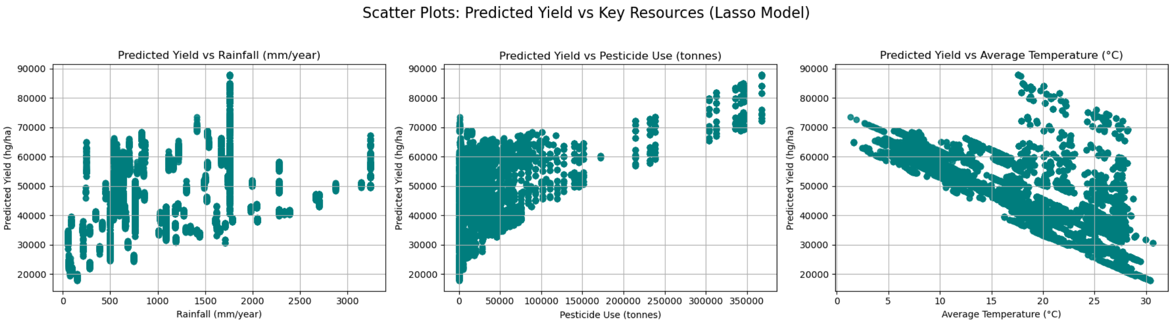
```
- Average Rainfall: Shows a positive relationship with
yield, indicating that increases in rainfall generally
lead to higher yields.
- Pesticides: Also displays a positive relationship,
suggesting that more pesticide use is associated with
greater crop yields.
- Average Temperature: Presents a negative relationship,
where higher temperatures correlate with lower yields,
likely due to heat stress on crops.
```

These insights help underline the importance of managing water and pesticide use effectively, while also considering the impacts of temperature on crop productivity.

Model Predicted Yield Sensitivity to Rainfall, Pesticide Use, and Temperature

In [37]:
```
# Start from unscaled input features
df_pred_plot = X_original.loc[X_train.index, ["average_rain_fall_mm_per_y

# Feature engineering
df_pred_plot["rainfall_temp_interaction"] = df_pred_plot["average_rain_fa
df_pred_plot["yield_per_rainfall"] = 1 / (df_pred_plot["average_rain_fall
df_pred_plot["yield_per_pesticide"] = 1 / (df_pred_plot["pesticides_tonne

# Scale the 3 original numerical features
df_pred_plot_scaled = df_pred_plot.copy()
df_pred_plot_scaled[["average_rain_fall_mm_per_year", "pesticides_tonnes"
    df_pred_plot_scaled[["average_rain_fall_mm_per_year", "pesticides_ton
)
```

```
# Align with model input
df_model_input = pd.DataFrame(0, index=df_pred_plot_scaled.index, columns
for col in df_pred_plot_scaled.columns:
    if col in df_model_input.columns:
        df_model_input[col] = df_pred_plot_scaled[col]

# Predict yield using Lasso model
predicted_yield = best_lasso.predict(df_model_input)

# Plot all three scatter plots
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

features = [
    ("average_rain_fall_mm_per_year", "Rainfall (mm/year)"),
    ("pesticides_tonnes", "Pesticide Use (tonnes)"),
    ("avg_temp", "Average Temperature (°C)")
]

for ax, (feature, label) in zip(axes, features):
    ax.scatter(X_original.loc[X_train.index, feature], predicted_yield, a
    ax.set_xlabel(label)
    ax.set_ylabel("Predicted Yield (hg/ha)")
    ax.set_title(f"Predicted Yield vs {label}")
    ax.grid(True)

plt.suptitle("Scatter Plots: Predicted Yield vs Key Resources (Lasso Mode
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```



These scatter plots show the relationship between predicted crop yields and three key resources:

```
- Rainfall: The data shows a broad spread, indicating no
strong consistent trend between rainfall and yield. Higher
rainfall sometimes corresponds to higher yields, but the
wide variability at various rainfall levels suggests that
additional factors influence the yield beyond just changes
in rainfall.
- Pesticide Use: The data clusters at lower pesticide
levels with varying yields, indicating a potential
threshold beyond which additional pesticide use does not
significantly increase yield.
- Temperature: The relationship appears nonlinear, with
moderate temperatures correlating with higher yields,
while both low and high extremes are associated with lower
yields. This pattern suggests that there is an optimal
temperature range for maximizing crop yield, outside of
```

which yields tend to decline.

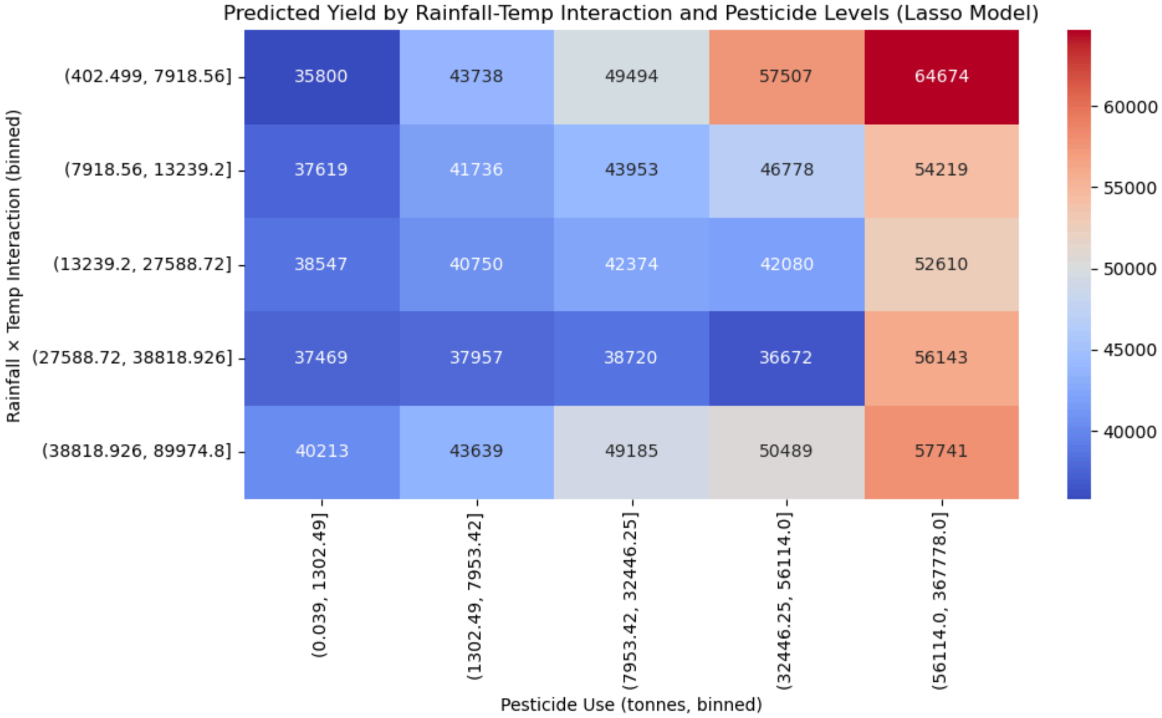
These visualizations highlight the complex dynamics of agricultural resources and their impact on yield, suggesting that maintaining resource levels within certain optimal ranges is crucial, and deviations from these ranges could adversely affect crop productivity.

These two types of plots highlight that while increases in rainfall and pesticide use generally correlate with higher yields according to the partial dependence plots, the scatter plots reveal more complex interactions and variability. This suggests that while these resources are important, their impact on yield is influenced by a multitude of other factors such as soil quality, crop types, and local climate conditions, which can significantly modify the observed relationships.

Yield Sensitivity to Combined Climate & Pesticide

In [39]:
```
# Start from unscaled original features
df_heatmap_interaction = X_original.loc[X_train.index, ["average_rain_fal

# Feature engineering (match model)
df_heatmap_interaction["rainfall_temp_interaction"] = df_heatmap_interact
df_heatmap_interaction["yield_per_rainfall"] = 1 / (df_heatmap_interactio
df_heatmap_interaction["yield_per_pesticide"] = 1 / (df_heatmap_interacti

# Scale original numerical features
df_scaled = df_heatmap_interaction.copy()
df_scaled[["average_rain_fall_mm_per_year", "pesticides_tonnes", "avg_tem
    df_scaled[["average_rain_fall_mm_per_year", "pesticides_tonnes", "avg
)

# Align with model input
df_input = pd.DataFrame(0, index=df_scaled.index, columns=X_train_importa
for col in df_scaled.columns:
    if col in df_input.columns:
        df_input[col] = df_scaled[col]

# Predict using Lasso
df_heatmap_interaction["Predicted Yield"] = best_lasso.predict(df_input)

# Bin rainfall-temp interaction and pesticide
df_heatmap_interaction["Interaction Bin"] = pd.qcut(df_heatmap_interactio
df_heatmap_interaction["Pesticide Bin"] = pd.qcut(df_heatmap_interaction[

# Pivot for heatmap
heatmap_interaction = df_heatmap_interaction.pivot_table(
    index="Interaction Bin",
    columns="Pesticide Bin",
    values="Predicted Yield",
    aggfunc="mean",
    observed=False
)

# Plot
plt.figure(figsize=(10, 6))
```

```
sns.heatmap(heatmap_interaction, annot=True, fmt=".0f", cmap="coolwarm")
plt.title("Predicted Yield by Rainfall-Temp Interaction and Pesticide Lev
plt.xlabel("Pesticide Use (tonnes, binned)")
plt.ylabel("Rainfall × Temp Interaction (binned)")
plt.tight_layout()
plt.show()
```



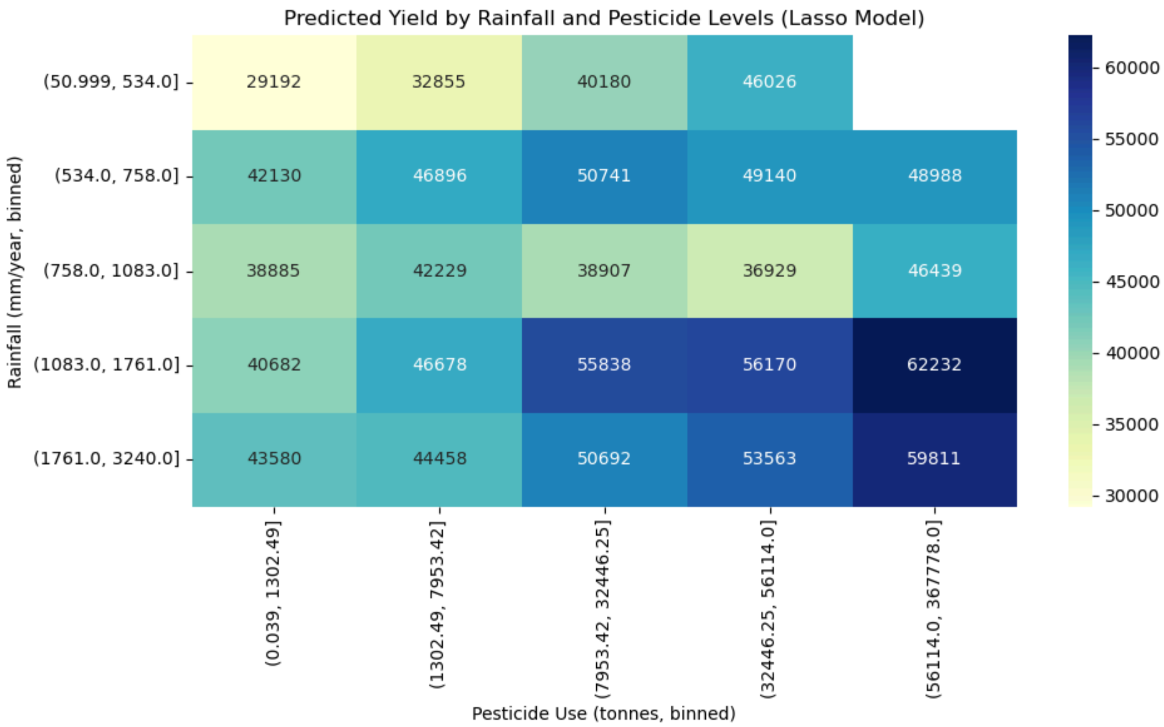
This heatmap visualizes the predicted yield from a Lasso Model based on the interaction between rainfall and temperature (binned) and varying levels of pesticide use. Yields generally increase with higher pesticide levels across most rainfall-temperature interactions. Note that the highest yields are observed with the highest pesticide use in moderate to high rainfall-temperature interaction bins. Conversely, lower pesticide levels tend to correspond with the lowest yields, particularly in higher interaction bins, suggesting that optimal yields are achieved with higher pesticide inputs in environments with significant climate interaction.

Yield Prediction based on Rainfall and Pesticide

In [41]:

```
# Step 1: Copy original unscaled features from training set
df_heatmap = X_original.loc[X_train.index, ["average_rain_fall_mm_per_yea

# Step 2: Feature Engineering (same as training)
df_heatmap["rainfall_temp_interaction"] = df_heatmap["average_rain_fall_m
df_heatmap["yield_per_rainfall"] = 1 / (df_heatmap["average_rain_fall_mm_
df_heatmap["yield_per_pesticide"] = 1 / (df_heatmap["pesticides_tonnes"]

# Step 3: Scale only the original 3 features
df_heatmap_scaled = df_heatmap.copy()
df_heatmap_scaled[["average_rain_fall_mm_per_year", "pesticides_tonnes",
    df_heatmap_scaled[["average_rain_fall_mm_per_year", "pesticides_tonne
)

# Step 4: Align columns with Lasso input (X_train_important.columns)
df_heatmap_model_input = pd.DataFrame(0, index=df_heatmap_scaled.index, c
```

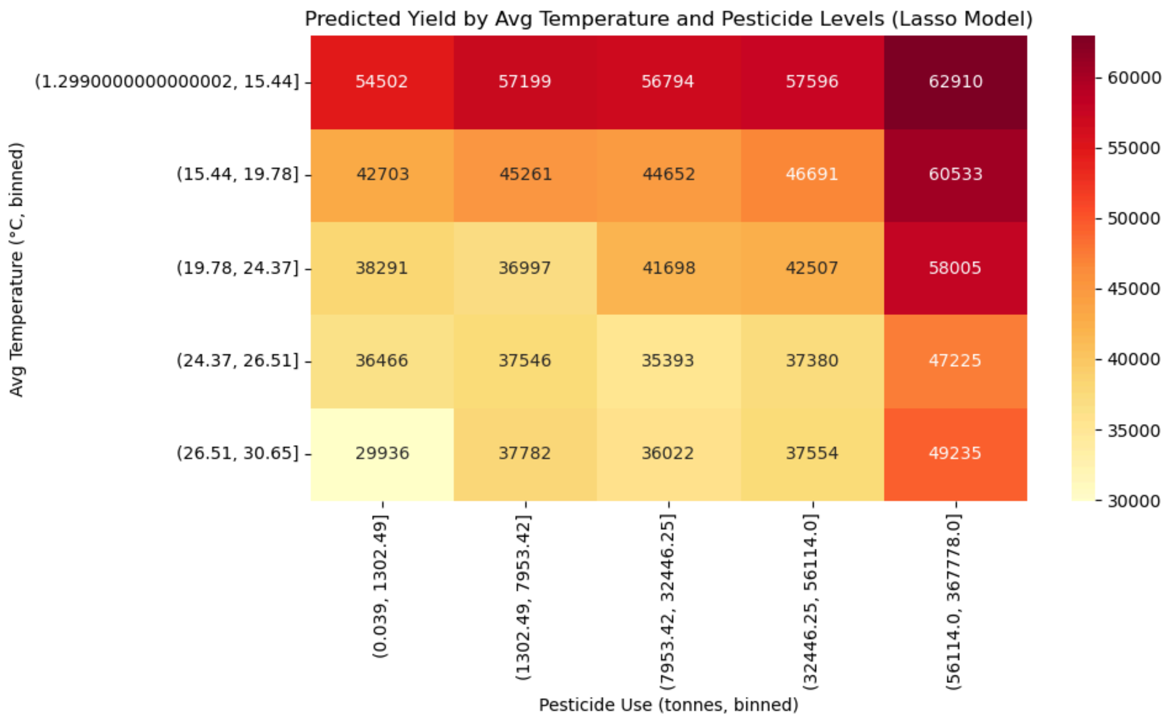
```
for col in df_heatmap_scaled.columns:
    if col in df_heatmap_model_input.columns:
        df_heatmap_model_input[col] = df_heatmap_scaled[col]

# Step 5: Predict yield using best Lasso model
df_heatmap["Predicted Yield"] = best_lasso.predict(df_heatmap_model_input

# Step 6: Bin rainfall and pesticide for plotting
df_heatmap["Rainfall Bin"] = pd.qcut(df_heatmap["average_rain_fall_mm_per
df_heatmap["Pesticide Bin"] = pd.qcut(df_heatmap["pesticides_tonnes"], q=

# Step 7: Pivot table for heatmap
heatmap_predicted = df_heatmap.pivot_table(
    index="Rainfall Bin",
    columns="Pesticide Bin",
    values="Predicted Yield",
    aggfunc="mean",
    observed=False
)

# Step 8: Plot
plt.figure(figsize=(10, 6))
sns.heatmap(heatmap_predicted, annot=True, fmt=".0f", cmap="YlGnBu")
plt.title("Predicted Yield by Rainfall and Pesticide Levels (Lasso Model)
plt.xlabel("Pesticide Use (tonnes, binned)")
plt.ylabel("Rainfall (mm/year, binned)")
plt.tight_layout()
plt.show()
```



This heatmap shows the predicted yield from a Lasso Model based on varying levels of rainfall and pesticide use. As rainfall increases, there is a general trend of increasing yield across most pesticide levels. However, the highest yields are observed with the highest pesticide use, particularly in the high rainfall bins. This suggests that optimal yields are achieved when higher rainfall is complemented by substantial pesticide application, indicating that both factors play critical roles in maximizing agricultural output.

Yield Prediction based on Average Temperature and Pesticide

In [43]:
```
# Copy original unscaled features
df_heatmap_temp = X_original.loc[X_train.index, ["average_rain_fall_mm_pe

# Feature Engineering (same as training)
df_heatmap_temp["rainfall_temp_interaction"] = df_heatmap_temp["average_r
df_heatmap_temp["yield_per_rainfall"] = 1 / (df_heatmap_temp["average_rai
df_heatmap_temp["yield_per_pesticide"] = 1 / (df_heatmap_temp["pesticides

# Scale only the original 3 features
df_heatmap_scaled_temp = df_heatmap_temp.copy()
df_heatmap_scaled_temp[["average_rain_fall_mm_per_year", "pesticides_tonn
    df_heatmap_scaled_temp[["average_rain_fall_mm_per_year", "pesticides_
)

# Align with Lasso model input
df_input_temp = pd.DataFrame(0, index=df_heatmap_scaled_temp.index, colum
for col in df_heatmap_scaled_temp.columns:
    if col in df_input_temp.columns:
        df_input_temp[col] = df_heatmap_scaled_temp[col]

# Predict
df_heatmap_temp["Predicted Yield"] = best_lasso.predict(df_input_temp)

# Bin avg_temp and pesticides for heatmap axes
df_heatmap_temp["Temp Bin"] = pd.qcut(df_heatmap_temp["avg_temp"], q=5)
df_heatmap_temp["Pesticide Bin"] = pd.qcut(df_heatmap_temp["pesticides_to

# Pivot for heatmap
heatmap_temp_pest = df_heatmap_temp.pivot_table(
    index="Temp Bin",
    columns="Pesticide Bin",
    values="Predicted Yield",
    aggfunc="mean",
    observed=False
)

# Plot heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(heatmap_temp_pest, annot=True, fmt=".0f", cmap="YlOrRd")
plt.title("Predicted Yield by Avg Temperature and Pesticide Levels (Lasso
plt.xlabel("Pesticide Use (tonnes, binned)")
plt.ylabel("Avg Temperature (°C, binned)")
plt.tight_layout()
plt.show()
```



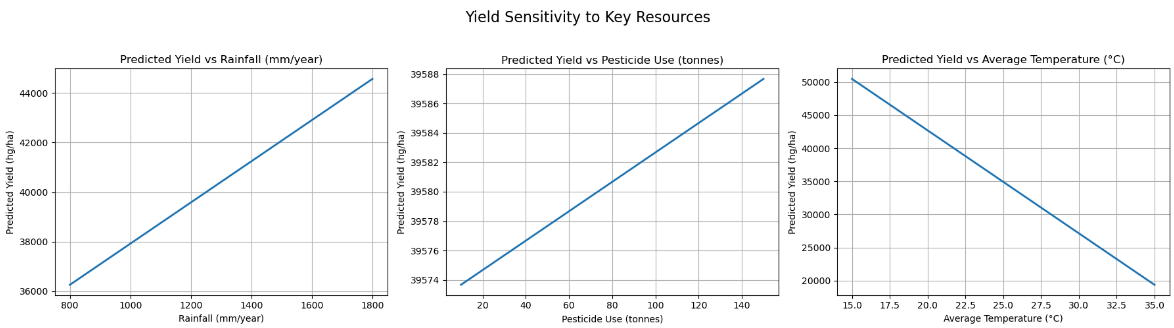
This heatmap depicts the predicted crop yields based on average temperature and pesticide use levels according to a Lasso Model. Yields generally decrease as temperatures increase, particularly in higher pesticide use bins. The highest yields are achieved at cooler temperatures, regardless of pesticide levels, indicating temperature is a critical factor influencing yield. The lowest yields occur at the highest temperature range across all pesticide levels, showing the adverse effects of high temperatures on crop productivity.

Predicted Yield Sensitivity to Key Agricultural Inputs

In [45]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Constants for fixing other variables
fixed_rainfall = 1200
fixed_pesticide = 100
fixed_temp = 22

# Define feature ranges and correct mapping
scenarios = {
    "Rainfall (mm/year)": ("average_rain_fall_mm_per_year", np.linspace(8
    "Pesticide Use (tonnes)": ("pesticides_tonnes", np.linspace(10, 150,
    "Average Temperature (°C)": ("avg_temp", np.linspace(15, 35, 50))
}

# Create base DataFrames
predictions = {}

for label, (feature_name, feature_range) in scenarios.items():
    df = pd.DataFrame({
        "average_rain_fall_mm_per_year": fixed_rainfall,
        "pesticides_tonnes": fixed_pesticide,
        "avg_temp": fixed_temp
```

```
    }, index=range(len(feature_range)))

    df[feature_name] = feature_range

    # Feature engineering
    df["rainfall_temp_interaction"] = df["average_rain_fall_mm_per_year"]
    df["yield_per_rainfall"] = 1 / (df["average_rain_fall_mm_per_year"] +
    df["yield_per_pesticide"] = 1 / (df["pesticides_tonnes"] + 1e-5)

    # Scale original features
    df_scaled = df.copy()
    df_scaled[["average_rain_fall_mm_per_year", "pesticides_tonnes", "avg
        df_scaled[["average_rain_fall_mm_per_year", "pesticides_tonnes",
    )

    # Align with model input
    df_input = pd.DataFrame(0, index=df_scaled.index, columns=X_train_imp
    for col in df_scaled.columns:
        if col in df_input.columns:
            df_input[col] = df_scaled[col]

    # Predict
    df["Predicted Yield"] = best_lasso.predict(df_input)

    # Store for plotting
    predictions[label] = (feature_range, df["Predicted Yield"])

# --- Plotting ---
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

for ax, (label, (x_vals, y_vals)) in zip(axes, predictions.items()):
    ax.plot(x_vals, y_vals, lw=2)
    ax.set_title(f"Predicted Yield vs {label}")
    ax.set_xlabel(label)
    ax.set_ylabel("Predicted Yield (hg/ha)")
    ax.grid(True)

plt.suptitle("Yield Sensitivity to Key Resources", fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```



These line graphs illustrate the sensitivity of predicted crop yields to three key variables:

```
- Rainfall: Shows a strong positive correlation; as
rainfall increases, predicted yield consistently rises,
indicating the beneficial impact of water availability on
crop productivity.
- Pesticide Use: Similarly, there is a positive
```

```
relationship between the amount of pesticide used and the
predicted yield, suggesting that higher pesticide usage
effectively enhances crop yields.
- Average Temperature: Displays a negative correlation; as
temperature increases, predicted yield decreases,
highlighting the adverse effects of higher temperatures on
crop production.
```

These trends show the importance of managing water, pesticide use, and adapting to temperature variations to optimize agricultural outputs.

Predict Crop Yield for New Data

In [47]:
```
# New data points for prediction
new_data = [
    {"Year": 2023, "average_rain_fall_mm_per_year": 1500, "pesticides_ton
    {"Year": 2024, "average_rain_fall_mm_per_year": 1200, "pesticides_ton
    {"Year": 2025, "average_rain_fall_mm_per_year": 1700, "pesticides_ton
    {"Year": 2026, "average_rain_fall_mm_per_year": 1000, "pesticides_ton
]

# Convert to DataFrame
new_df = pd.DataFrame(new_data)

# Step 1: Feature Engineering (same as training)
new_df["rainfall_temp_interaction"] = new_df["average_rain_fall_mm_per_ye
new_df["yield_per_rainfall"] = 1 / (new_df["average_rain_fall_mm_per_year
new_df["yield_per_pesticide"] = 1 / (new_df["pesticides_tonnes"] + 1e-5)

# Step 2: Scale only the original 3 features
new_df[["average_rain_fall_mm_per_year", "pesticides_tonnes", "avg_temp"]
    new_df[["average_rain_fall_mm_per_year", "pesticides_tonnes", "avg_te
)

# Step 3: Prepare aligned DataFrame with all expected columns (from X_tra
new_df_aligned = pd.DataFrame(0, index=range(len(new_df)), columns=X_trai

# Fill values for any matching columns
for col in new_df.columns:
    if col in new_df_aligned.columns:
        new_df_aligned[col] = new_df[col]

# Step 4: Predict using trained Lasso model
predicted_yield_lasso = best_lasso.predict(new_df_aligned)

# Step 5: Add predictions to the original DataFrame
new_df["Predicted_Yield_Lasso (hg/ha)"] = predicted_yield_lasso

# Step 6: Display results
print("\nPredicted Yields for New Scenarios (Lasso):")
print(new_df[["Year", "Predicted_Yield_Lasso (hg/ha)"]].to_string(index=F
```

```
Predicted Yields for New Scenarios (Lasso):
 Year  Predicted_Yield_Lasso (hg/ha)
 2023                   45181.956208
 2024                   39580.669501
 2025                   49952.899931
 2026                   33254.596899
```

The predicted yields for new crop scenarios using a trained Lasso model indicate varied outputs based on different conditions of rainfall, pesticide use, and temperature across several locations and crop types. Here are the results:

```
- 2023, Maize in Albania: Predicted yield is approximately
45,182 hg/ha, influenced by favorable rainfall and
moderate pesticide use.
- 2024, Wheat in India: Yield is predicted at about 39,581
hg/ha, with somewhat lower rainfall and pesticide use.
- 2025, Rice in the USA: High rainfall and pesticide use
result in a predicted yield of roughly 49,953 hg/ha.
- 2026, Barley in Canada: The lowest yield at
approximately 33,255 hg/ha, attributed to the least
rainfall and high temperatures.
```

These predictions reflect how critical environmental factors and agronomic practices impact agricultural productivity, helping guide optimal resource allocation for each scenario.

Part II: Yield Prediction and Risk Mitigation

Importing libraries and loading the dataset

In [123…

```
# making shell calls directly into this cell
!pip install tensorflow
!pip install keras
```

```
Requirement already satisfied: tensorflow in /opt/anaconda3/lib/python3.1
2/site-packages (2.17.0)
Requirement already satisfied: absl-py>=1.0.0 in /opt/anaconda3/lib/python
3.12/site-packages (from tensorflow) (2.1.0)
Requirement already satisfied: astunparse>=1.6.0 in /opt/anaconda3/lib/pyt
hon3.12/site-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in /opt/anaconda3/lib/
python3.12/site-packages (from tensorflow) (24.3.25)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /op
t/anaconda3/lib/python3.12/site-packages (from tensorflow) (0.5.3)
Requirement already satisfied: google-pasta>=0.1.1 in /opt/anaconda3/lib/p
ython3.12/site-packages (from tensorflow) (0.2.0)
Requirement already satisfied: h5py>=3.10.0 in /opt/anaconda3/lib/python3.
12/site-packages (from tensorflow) (3.11.0)
Requirement already satisfied: ml-dtypes<0.5.0,>=0.3.1 in /opt/anaconda3/l
ib/python3.12/site-packages (from tensorflow) (0.4.0)
Requirement already satisfied: opt-einsum>=2.3.2 in /opt/anaconda3/lib/pyt
hon3.12/site-packages (from tensorflow) (3.3.0)
Requirement already satisfied: packaging in /opt/anaconda3/lib/python3.12/
site-packages (from tensorflow) (24.1)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.
3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3 in /opt/anaconda3/lib/python3.12/si
te-packages (from tensorflow) (4.25.3)
Requirement already satisfied: requests<3,>=2.21.0 in /opt/anaconda3/lib/p
ython3.12/site-packages (from tensorflow) (2.32.3)
Requirement already satisfied: setuptools in /opt/anaconda3/lib/python3.1
2/site-packages (from tensorflow) (75.1.0)
Requirement already satisfied: six>=1.12.0 in /opt/anaconda3/lib/python3.1
2/site-packages (from tensorflow) (1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in /opt/anaconda3/lib/pyth
on3.12/site-packages (from tensorflow) (2.1.0)
Requirement already satisfied: typing-extensions>=3.6.6 in /opt/anaconda3/
lib/python3.12/site-packages (from tensorflow) (4.11.0)
Requirement already satisfied: wrapt>=1.11.0 in /opt/anaconda3/lib/python
3.12/site-packages (from tensorflow) (1.14.1)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /opt/anaconda3/lib/p
ython3.12/site-packages (from tensorflow) (1.62.2)
Requirement already satisfied: tensorboard<2.18,>=2.17 in /opt/anaconda3/l
ib/python3.12/site-packages (from tensorflow) (2.17.0)
Requirement already satisfied: keras>=3.2.0 in /opt/anaconda3/lib/python3.
12/site-packages (from tensorflow) (3.6.0)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /opt/anaconda3/lib/py
thon3.12/site-packages (from astunparse>=1.6.0->tensorflow) (0.44.0)
Requirement already satisfied: numpy>=1.17.3 in /opt/anaconda3/lib/python
3.12/site-packages (from h5py>=3.10.0->tensorflow) (1.26.4)
Requirement already satisfied: rich in /opt/anaconda3/lib/python3.12/site-
packages (from keras>=3.2.0->tensorflow) (13.7.1)
Requirement already satisfied: namex in /opt/anaconda3/lib/python3.12/site
-packages (from keras>=3.2.0->tensorflow) (0.0.7)
Requirement already satisfied: optree in /opt/anaconda3/lib/python3.12/sit
e-packages (from keras>=3.2.0->tensorflow) (0.12.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /opt/anaconda3/
lib/python3.12/site-packages (from requests<3,>=2.21.0->tensorflow) (3.3.
2)
Requirement already satisfied: idna<4,>=2.5 in /opt/anaconda3/lib/python3.
12/site-packages (from requests<3,>=2.21.0->tensorflow) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /opt/anaconda3/lib/py
thon3.12/site-packages (from requests<3,>=2.21.0->tensorflow) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in /opt/anaconda3/lib/py
thon3.12/site-packages (from requests<3,>=2.21.0->tensorflow) (2025.1.31)
```

```
Requirement already satisfied: markdown>=2.6.8 in /opt/anaconda3/lib/pytho
n3.12/site-packages (from tensorboard<2.18,>=2.17->tensorflow) (3.4.1)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /o
pt/anaconda3/lib/python3.12/site-packages (from tensorboard<2.18,>=2.17->t
ensorflow) (0.7.0)
Requirement already satisfied: werkzeug>=1.0.1 in /opt/anaconda3/lib/pytho
n3.12/site-packages (from tensorboard<2.18,>=2.17->tensorflow) (3.0.3)
Requirement already satisfied: MarkupSafe>=2.1.1 in /opt/anaconda3/lib/pyt
hon3.12/site-packages (from werkzeug>=1.0.1->tensorboard<2.18,>=2.17->tens
orflow) (2.1.3)
Requirement already satisfied: markdown-it-py>=2.2.0 in /opt/anaconda3/li
b/python3.12/site-packages (from rich->keras>=3.2.0->tensorflow) (2.2.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /opt/anaconda3/l
ib/python3.12/site-packages (from rich->keras>=3.2.0->tensorflow) (2.15.1)
Requirement already satisfied: mdurl~=0.1 in /opt/anaconda3/lib/python3.1
2/site-packages (from markdown-it-py>=2.2.0->rich->keras>=3.2.0->tensorflo
w) (0.1.0)
Requirement already satisfied: keras in /opt/anaconda3/lib/python3.12/site
-packages (3.6.0)
Requirement already satisfied: absl-py in /opt/anaconda3/lib/python3.12/si
te-packages (from keras) (2.1.0)
Requirement already satisfied: numpy in /opt/anaconda3/lib/python3.12/site
-packages (from keras) (1.26.4)
Requirement already satisfied: rich in /opt/anaconda3/lib/python3.12/site-
packages (from keras) (13.7.1)
Requirement already satisfied: namex in /opt/anaconda3/lib/python3.12/site
-packages (from keras) (0.0.7)
Requirement already satisfied: h5py in /opt/anaconda3/lib/python3.12/site-
packages (from keras) (3.11.0)
Requirement already satisfied: optree in /opt/anaconda3/lib/python3.12/sit
e-packages (from keras) (0.12.1)
Requirement already satisfied: ml-dtypes in /opt/anaconda3/lib/python3.12/
site-packages (from keras) (0.4.0)
Requirement already satisfied: packaging in /opt/anaconda3/lib/python3.12/
site-packages (from keras) (24.1)
Requirement already satisfied: typing-extensions>=4.5.0 in /opt/anaconda3/
lib/python3.12/site-packages (from optree->keras) (4.11.0)
Requirement already satisfied: markdown-it-py>=2.2.0 in /opt/anaconda3/li
b/python3.12/site-packages (from rich->keras) (2.2.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /opt/anaconda3/l
ib/python3.12/site-packages (from rich->keras) (2.15.1)
Requirement already satisfied: mdurl~=0.1 in /opt/anaconda3/lib/python3.1
2/site-packages (from markdown-it-py>=2.2.0->rich->keras) (0.1.0)
```

In [125…

```python
# import standard libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# import model libraries
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import RobustScaler
from sklearn.model_selection import GridSearchCV
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout, Input
from keras.optimizers import Adam
```

```python
# loading the dataset
yield_df = pd.read_csv("yield_df.csv")
```

Creating indicator variables for transformations

If the exponential or log transformations are applied repeatedly (which can happen unknowingly while re-running code blocks), the data values quickly shoot to infinity or drop down to 0 (or NaN). To prevent this, we added flags for each transformation. The transformations are applied only if the flags are 'False', ensuring that the transformation are not applied repeatedly.

In [127…

```python
yield_df_target_log = False
y_pred_lstm_original_exp = False
y_test_original_exp = False
yield_df_target_exp = False
yield_df_features_exp = False
x_scaled_t = False
y_scaled_t = False
```

This indicates that there are several outliers in the dataset. This can also have negative effects on models. We will therefore have to trim the data to remove most of these outliers.

Feature Engineering - Add derived features

Currently, the dataset has only 3 features - avg rainfall, pesticide usage, and avg temp. We will add some more features to make sure the model is able to understand the deeper depencies and is able to identify the underlying patterns and trends.

In [129…

```python
# create average rolling values for rainfall and temperature
yield_df = yield_df.sort_values(by=['Area', 'Item', 'Year'])
yield_df['rainfall_rolling_avg'] = yield_df.groupby(['Area', 'Item'], obs
yield_df['temp_rolling_avg'] = yield_df.groupby(['Area', 'Item'], observe

print(yield_df[['Area', 'Item', 'Year', 'average_rain_fall_mm_per_year',
```

```
       Area   Item  Year  average_rain_fall_mm_per_year  rainfall_rolling_
avg  \
0   Albania  Maize  1990                         1485.0               148
5.0
6   Albania  Maize  1991                         1485.0               148
5.0
12  Albania  Maize  1992                         1485.0               148
5.0
18  Albania  Maize  1993                         1485.0               148
5.0
23  Albania  Maize  1994                         1485.0               148
5.0
27  Albania  Maize  1995                         1485.0               148
5.0
31  Albania  Maize  1996                         1485.0               148
5.0
35  Albania  Maize  1997                         1485.0               148
5.0
39  Albania  Maize  1998                         1485.0               148
5.0
43  Albania  Maize  1999                         1485.0               148
5.0

    avg_temp  temp_rolling_avg
0      16.37         16.370000
6      15.36         15.865000
12     16.06         15.930000
18     16.05         15.823333
23     16.96         16.356667
27     15.67         16.226667
31     15.64         16.090000
35     15.90         15.736667
39     16.27         15.936667
43     16.57         16.246667
```

In [131…

```
# Calculate year-over-year changes for rainfall and temperature
yield_df['rainfall_yearly_change'] = yield_df.groupby(['Area', 'Item'], o
yield_df['temp_yearly_change'] = yield_df.groupby(['Area', 'Item'], obser

print(yield_df[['Area', 'Item', 'Year', 'rainfall_yearly_change', 'temp_y
```

```
       Area   Item  Year  rainfall_yearly_change  temp_yearly_change
0   Albania  Maize  1990                     NaN                 NaN
6   Albania  Maize  1991                     0.0               -1.01
12  Albania  Maize  1992                     0.0                0.70
18  Albania  Maize  1993                     0.0               -0.01
23  Albania  Maize  1994                     0.0                0.91
27  Albania  Maize  1995                     0.0               -1.29
31  Albania  Maize  1996                     0.0               -0.03
35  Albania  Maize  1997                     0.0                0.26
39  Albania  Maize  1998                     0.0                0.37
43  Albania  Maize  1999                     0.0                0.30
```

In [133…

```
# Create lag features for rainfall and temperature (previous year)
yield_df['rainfall_lag_1'] = yield_df.groupby(['Area', 'Item'], observed
yield_df['temp_lag_1'] = yield_df.groupby(['Area', 'Item'], observed = Fa

print(yield_df[['Area', 'Item', 'Year', 'rainfall_lag_1', 'temp_lag_1']].
```

```
       Area   Item  Year  rainfall_lag_1  temp_lag_1
0   Albania  Maize  1990             NaN         NaN
6   Albania  Maize  1991          1485.0       16.37
12  Albania  Maize  1992          1485.0       15.36
18  Albania  Maize  1993          1485.0       16.06
23  Albania  Maize  1994          1485.0       16.05
27  Albania  Maize  1995          1485.0       16.96
31  Albania  Maize  1996          1485.0       15.67
35  Albania  Maize  1997          1485.0       15.64
39  Albania  Maize  1998          1485.0       15.90
43  Albania  Maize  1999          1485.0       16.27
```

Define the target variable and features

In [135…

```
# set target variable
target = 'hg/ha_yield'
features = ['average_rain_fall_mm_per_year', 'pesticides_tonnes',
            'avg_temp', 'rainfall_rolling_avg', 'temp_rolling_avg',
            'rainfall_yearly_change', 'temp_yearly_change', 'rainfall_lag
            'temp_lag_1']
```

Data Preprocessing

Adding new features can introduces some missing data points. We remove this using the `bfill()` and `ffill()` methods.

In [137…

```
yield_df[features] = yield_df[features].ffill()
yield_df[features] = yield_df[features].bfill()
```

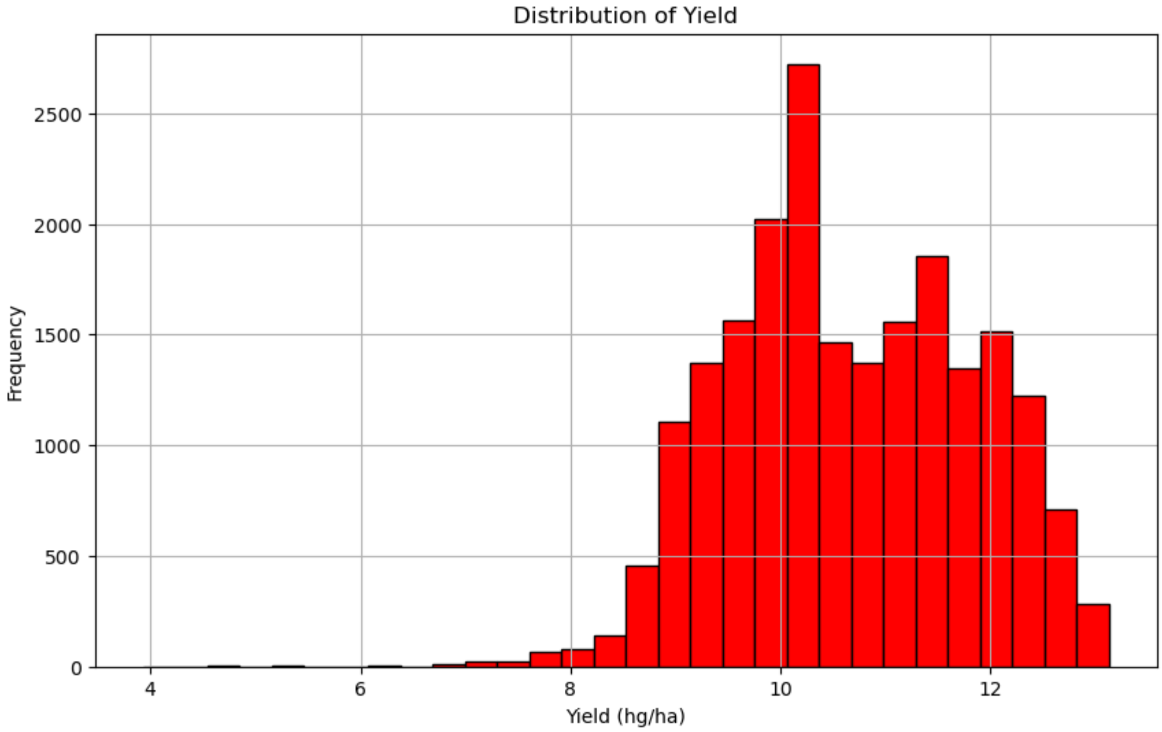
Since there are a lot of outliers in the dataset, we will trim the dataset to remove some of them.

We will also transform the target variable using a log transformation to remove the heavy leftwards-skewness.

In [139…

```
def preprocess_data(df):
    # Remove extreme outliers
    for feature in features:
        Q1 = df[feature].quantile(0.25)
        Q3 = df[feature].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        df = df[(df[feature] >= lower_bound) & (df[feature] <= upper_boun

    # Log transform the target variable
    if (yield_df_target_log == False):
      df[target] = np.log1p(df[target])
      log_t = True


    return df

yield_df = preprocess_data(yield_df)
```

In [141…
```
# visualize distribution of target variable
plt.figure(figsize=(10,6))
yield_df[target].hist(bins=30, color='red', edgecolor='black')
plt.title('Distribution of Yield')
plt.xlabel('Yield (hg/ha)')
plt.ylabel('Frequency')
plt.show()
```



As you can see, the target variable is no longer left-skewed. This makes is easier to work with it.

Model 1: Random Forest Regressor

Aim: Extract the most important features

In [143…
```
# split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(yield_df[features], y
```

We are using GridSearchCV to find the optimal values for the model parameters.

In [333…
```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [100, 200, 500],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

grid_search = GridSearchCV(estimator=RandomForestRegressor(random_state=4
grid_search.fit(X_train, y_train)

print("Best parameters:", grid_search.best_params_)
```

```
Best parameters: {'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_spl
it': 5, 'n_estimators': 200}
```

Since there are multiple features and a big training set, the `grid_search()` function takes about 35 minutes to run. We are therefore saving the parameter values below, to save time.

In [145…
```
# saving best params from grid search
best_params = {'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split
print(best_params)
```

```
{'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 5, 'n_estima
tors': 200}
```

In [147…
```
# flatten X_train and X_test (for debugging)
X_train_flat = np.mean(X_train, axis=1)
X_test_flat = np.mean(X_test, axis=1)
```

In [149…
```
# train the Random Forest Regressor
rf_model = RandomForestRegressor(n_estimators=200, random_state=42, max_d
rf_model.fit(X_train, y_train)
```

Out[149…
```
RandomForestRegressor

RandomForestRegressor(max_depth=10, min_samples_leaf=2, min_sampl
es_split=5,
                      n_estimators=200, random_state=42)
```

In [151…
```
# extract important features
rf_model.feature_importances_
importance_df = pd.DataFrame({'feature': features, 'importance': rf_model
importance_df.sort_values('importance', ascending=False)
```

Out[151…

	feature	importance
1	pesticides_tonnes	0.288272
4	temp_rolling_avg	0.212972
3	rainfall_rolling_avg	0.098875
2	avg_temp	0.094030
0	average_rain_fall_mm_per_year	0.093186
7	rainfall_lag_1	0.081683
8	temp_lag_1	0.076417
6	temp_yearly_change	0.054563
5	rainfall_yearly_change	0.000000

We will use features with `importance >= 0.09`.

Model 2: LSTM

Aim: Use temporal nature of the data for accurate predictions

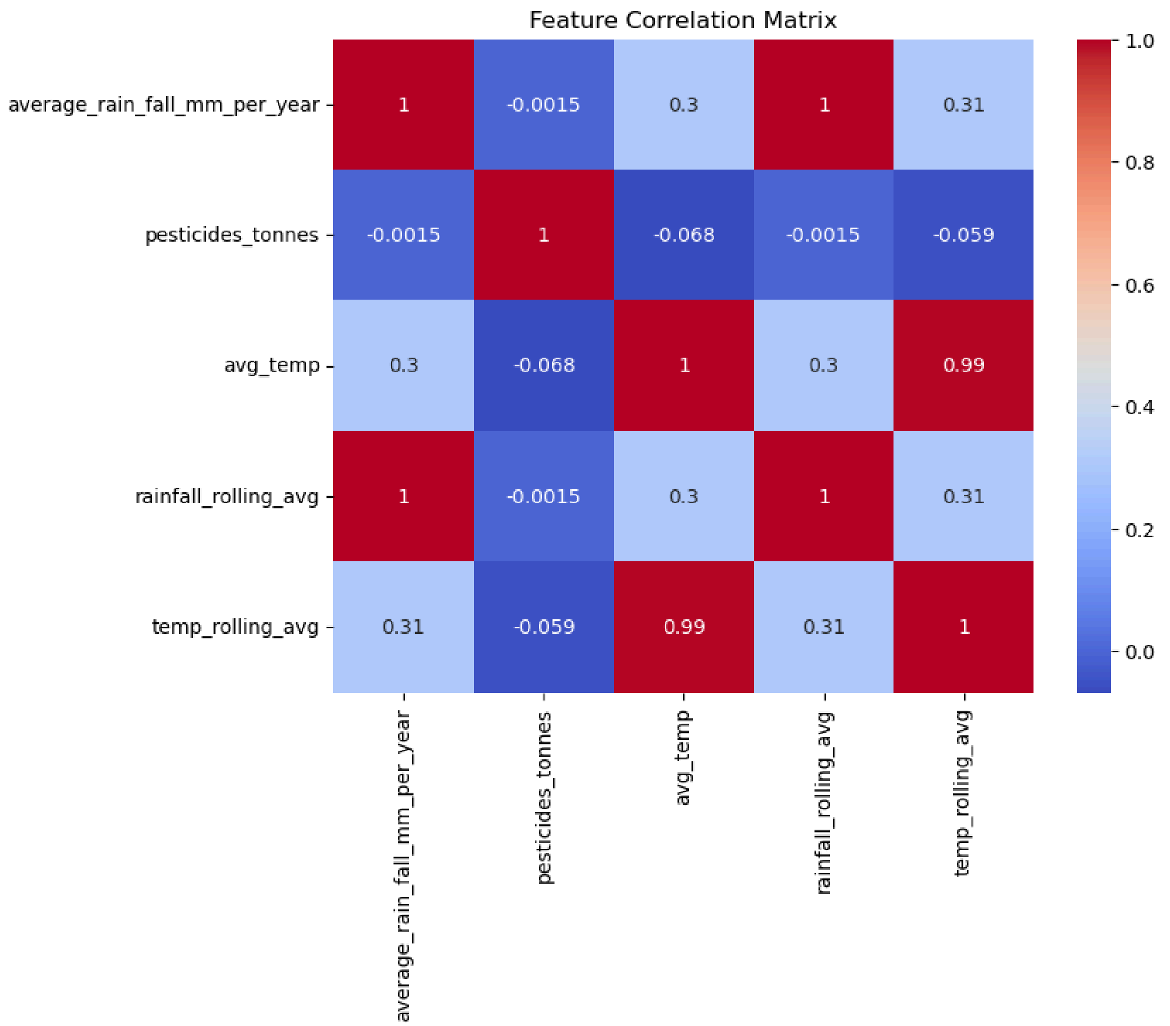
In [153…

```
# select features from random forest with confidence more than 0.09
threshold = 0.09  # Minimum importance score
top_features = importance_df[importance_df['importance'] > threshold]['fe

print(f"Features Selected Above Threshold ({threshold}): {top_features}")
```

```
Features Selected Above Threshold (0.09): ['average_rain_fall_mm_per_yea
r', 'pesticides_tonnes', 'avg_temp', 'rainfall_rolling_avg', 'temp_rolling
_avg']
```

In [155…

```
# compute correlation matrix
correlation_matrix = yield_df[top_features].corr()

# visualize correlation matrix
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Feature Correlation Matrix')
plt.show()
```



Let us quickly understand how LSTM's work.

- LSTM models are specifically designed for sequential data (like what we have here). They learn from past time steps to predict future ones.
- Sequences are ordered collections of data points. The 'order' matters in such models because of the temporal nature of the dataset.

- If we fed the entire dataset at once, the LSTM wouldn't know what's past and what's future.
- Instead, we break the dataset into sequences so that the model can learn from past trends and predict the next value.
- In our LSTM model below, we set the `sequence_length` or `seq_length` to 5.
- This means that the model will place emphasis on the data from the last 5 years to predict the yield for the next year.
- This helps the model capture recent trends like weather patterns, soil quality changes, or farming techniques.

In [157…

```
# create sequences for LSTM
def create_sequences_with_indices(data, seq_length):
    sequences = []
    # we need original indices to map the sequences back to the original
    # for risk_score prediction
    original_indices = []

    for i in range(len(data) - seq_length):
        # extract sequence of features
        sequence = data[i:i+seq_length, :-1]  # all columns except the la

        # extract label (target value)
        label = data[i+seq_length, -1]  # last column is the target

        # store original_index
        original_index = i + seq_length

        # append sequence and index
        sequences.append((sequence, label))
        original_indices.append(original_index)

    return sequences, original_indices

seq_length = 5  # no. of years in each sequence

# prepare data for sequence creation
data = yield_df[top_features + [target]].values  # include both features

# group data by Area and Item (Crop Type)
grouped_data = yield_df.groupby(['Area', 'Item'], observed=False)
sequences = []
all_original_indices = []

for _, group in grouped_data:
    group_sequences, group_indices = create_sequences_with_indices(group[
    sequences.extend(group_sequences)
    all_original_indices.extend(group_indices)
```

In [159…

```
# zip function unzips a list of tuples into seperate lists
# sequences is a list of tuples of (features, labels)
X, y = zip(*sequences)
X = np.array(X) # features
y = np.array(y) # labels
```

In [161…

```
# split the sequences into training, validation and testing sets

X_train, X_temp, y_train, y_temp, train_indices, temp_indices = train_tes

X_val, X_test, y_val, y_test, val_indices, test_indices = train_test_spli

print(f"Training data shape: {X_train.shape}, Validation data shape: {X_v
```

```
Training data shape: (10866, 5, 5), Validation data shape: (3622, 5, 5), T
esting data shape: (3622, 5, 5)
```

In [163…

```
# define lstm model
model = Sequential([
    Input(shape=(X_train.shape[1], X_train.shape[2])),
    LSTM(64, activation='tanh', return_sequences=True),
    Dropout(0.2),
    LSTM(32, activation='tanh', return_sequences=False),
    Dropout(0.2),
    Dense(1, activation='relu')
])

# add optmizer and loss function
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='mean_squared_error',
    metrics=['mae', 'mse']
)

# train the model
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epo
```

```
Epoch 1/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 35.4942 - mae: 4.7046 - m
se: 35.4942 - val_loss: 1.3117 - val_mae: 0.9667 - val_mse: 1.3117
Epoch 2/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.2770 - mae: 1.2150 - ms
e: 2.2770 - val_loss: 1.2831 - val_mae: 0.9594 - val_mse: 1.2831
Epoch 3/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 2.2391 - mae: 1.2112 - ms
e: 2.2391 - val_loss: 1.2731 - val_mae: 0.9628 - val_mse: 1.2731
Epoch 4/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 2.2673 - mae: 1.2256 - ms
e: 2.2673 - val_loss: 1.2748 - val_mae: 0.9591 - val_mse: 1.2748
Epoch 5/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.1801 - mae: 1.1890 - ms
e: 2.1801 - val_loss: 1.2704 - val_mae: 0.9603 - val_mse: 1.2704
Epoch 6/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.1415 - mae: 1.1812 - ms
e: 2.1415 - val_loss: 1.2772 - val_mae: 0.9616 - val_mse: 1.2772
Epoch 7/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.1244 - mae: 1.1793 - ms
e: 2.1244 - val_loss: 1.2940 - val_mae: 0.9575 - val_mse: 1.2940
Epoch 8/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.0823 - mae: 1.1729 - ms
e: 2.0823 - val_loss: 1.2870 - val_mae: 0.9654 - val_mse: 1.2870
Epoch 9/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.1730 - mae: 1.2004 - ms
e: 2.1730 - val_loss: 1.2776 - val_mae: 0.9663 - val_mse: 1.2776
Epoch 10/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 2.0840 - mae: 1.1697 - ms
e: 2.0840 - val_loss: 1.2714 - val_mae: 0.9634 - val_mse: 1.2714
Epoch 11/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 2.1753 - mae: 1.2079 - ms
e: 2.1753 - val_loss: 1.2771 - val_mae: 0.9590 - val_mse: 1.2771
Epoch 12/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 2.0794 - mae: 1.1744 - ms
e: 2.0794 - val_loss: 1.2725 - val_mae: 0.9633 - val_mse: 1.2725
Epoch 13/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.1539 - mae: 1.1888 - ms
e: 2.1539 - val_loss: 1.2723 - val_mae: 0.9597 - val_mse: 1.2723
Epoch 14/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.0196 - mae: 1.1571 - ms
e: 2.0196 - val_loss: 1.2813 - val_mae: 0.9584 - val_mse: 1.2813
Epoch 15/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.0620 - mae: 1.1671 - ms
e: 2.0620 - val_loss: 1.2797 - val_mae: 0.9579 - val_mse: 1.2797
Epoch 16/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.9745 - mae: 1.1382 - ms
e: 1.9745 - val_loss: 1.2817 - val_mae: 0.9568 - val_mse: 1.2817
Epoch 17/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.0229 - mae: 1.1569 - ms
e: 2.0229 - val_loss: 1.2736 - val_mae: 0.9577 - val_mse: 1.2736
Epoch 18/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.0449 - mae: 1.1685 - ms
e: 2.0449 - val_loss: 1.2720 - val_mae: 0.9637 - val_mse: 1.2720
Epoch 19/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 2.0152 - mae: 1.1540 - ms
e: 2.0152 - val_loss: 1.2701 - val_mae: 0.9633 - val_mse: 1.2701
Epoch 20/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 2.0209 - mae: 1.1625 - ms
e: 2.0209 - val_loss: 1.2815 - val_mae: 0.9565 - val_mse: 1.2815
```

```
Epoch 21/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.9753 - mae: 1.1445 - ms
e: 1.9753 - val_loss: 1.2744 - val_mae: 0.9663 - val_mse: 1.2744
Epoch 22/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.9679 - mae: 1.1408 - ms
e: 1.9679 - val_loss: 1.2739 - val_mae: 0.9597 - val_mse: 1.2739
Epoch 23/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.9186 - mae: 1.1278 - ms
e: 1.9186 - val_loss: 1.2779 - val_mae: 0.9690 - val_mse: 1.2779
Epoch 24/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.9503 - mae: 1.1410 - ms
e: 1.9503 - val_loss: 1.2732 - val_mae: 0.9598 - val_mse: 1.2732
Epoch 25/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.8829 - mae: 1.1169 - ms
e: 1.8829 - val_loss: 1.2791 - val_mae: 0.9576 - val_mse: 1.2791
Epoch 26/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.9342 - mae: 1.1355 - ms
e: 1.9342 - val_loss: 1.2657 - val_mae: 0.9609 - val_mse: 1.2657
Epoch 27/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.8667 - mae: 1.1157 - ms
e: 1.8667 - val_loss: 1.2727 - val_mae: 0.9654 - val_mse: 1.2727
Epoch 28/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.8577 - mae: 1.1083 - ms
e: 1.8577 - val_loss: 1.2702 - val_mae: 0.9645 - val_mse: 1.2702
Epoch 29/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.8535 - mae: 1.1133 - ms
e: 1.8535 - val_loss: 1.2822 - val_mae: 0.9568 - val_mse: 1.2822
Epoch 30/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.8610 - mae: 1.1139 - ms
e: 1.8610 - val_loss: 1.3311 - val_mae: 0.9608 - val_mse: 1.3311
Epoch 31/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - loss: 1.7823 - mae: 1.0893 - ms
e: 1.7823 - val_loss: 1.2855 - val_mae: 0.9577 - val_mse: 1.2855
Epoch 32/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.7772 - mae: 1.0851 - ms
e: 1.7772 - val_loss: 1.2804 - val_mae: 0.9702 - val_mse: 1.2804
Epoch 33/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.8171 - mae: 1.1004 - ms
e: 1.8171 - val_loss: 1.2674 - val_mae: 0.9631 - val_mse: 1.2674
Epoch 34/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.8164 - mae: 1.1038 - ms
e: 1.8164 - val_loss: 1.2707 - val_mae: 0.9587 - val_mse: 1.2707
Epoch 35/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.8100 - mae: 1.0994 - ms
e: 1.8100 - val_loss: 1.2950 - val_mae: 0.9590 - val_mse: 1.2950
Epoch 36/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.8191 - mae: 1.1102 - ms
e: 1.8191 - val_loss: 1.2875 - val_mae: 0.9580 - val_mse: 1.2875
Epoch 37/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.8164 - mae: 1.1040 - ms
e: 1.8164 - val_loss: 1.2677 - val_mae: 0.9576 - val_mse: 1.2677
Epoch 38/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.7447 - mae: 1.0803 - ms
e: 1.7447 - val_loss: 1.2727 - val_mae: 0.9656 - val_mse: 1.2727
Epoch 39/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.7264 - mae: 1.0788 - ms
e: 1.7264 - val_loss: 1.2723 - val_mae: 0.9581 - val_mse: 1.2723
Epoch 40/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.7329 - mae: 1.0790 - ms
e: 1.7329 - val_loss: 1.2664 - val_mae: 0.9595 - val_mse: 1.2664
```

```
Epoch 41/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.7713 - mae: 1.0933 - ms
e: 1.7713 - val_loss: 1.2706 - val_mae: 0.9653 - val_mse: 1.2706
Epoch 42/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.7739 - mae: 1.0954 - ms
e: 1.7739 - val_loss: 1.2762 - val_mae: 0.9559 - val_mse: 1.2762
Epoch 43/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.7291 - mae: 1.0814 - ms
e: 1.7291 - val_loss: 1.2736 - val_mae: 0.9632 - val_mse: 1.2736
Epoch 44/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.7279 - mae: 1.0750 - ms
e: 1.7279 - val_loss: 1.2651 - val_mae: 0.9601 - val_mse: 1.2651
Epoch 45/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.7208 - mae: 1.0811 - ms
e: 1.7208 - val_loss: 1.2699 - val_mae: 0.9590 - val_mse: 1.2699
Epoch 46/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.6851 - mae: 1.0714 - ms
e: 1.6851 - val_loss: 1.2662 - val_mae: 0.9587 - val_mse: 1.2662
Epoch 47/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.6587 - mae: 1.0569 - ms
e: 1.6587 - val_loss: 1.2631 - val_mae: 0.9571 - val_mse: 1.2631
Epoch 48/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 1.6214 - mae: 1.0445 - ms
e: 1.6214 - val_loss: 1.2709 - val_mae: 0.9570 - val_mse: 1.2709
Epoch 49/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.6888 - mae: 1.0703 - ms
e: 1.6888 - val_loss: 1.2790 - val_mae: 0.9553 - val_mse: 1.2790
Epoch 50/50
340/340 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.6753 - mae: 1.0623 - ms
e: 1.6753 - val_loss: 1.2837 - val_mae: 0.9719 - val_mse: 1.2837
```

Analyzing model results

In [165…

```
# visualizing the loss
plt.figure(figsize=(10,6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()
```

In [167…

```
# evaluate model loss
y_pred_lstm = model.predict(X_test)
y_pred_lstm_original = y_pred_lstm
y_test_original = y_test

# properly reverse transformations
if (y_pred_lstm_original_exp == False):
  y_pred_lstm_original = np.expm1(y_pred_lstm)
  y_pred_lstm_original_exp = True

if (y_test_original_exp == False):
  y_test_original = np.expm1(y_test)
  y_test_original_exp = True

mse_lstm = mean_squared_error(y_test, y_pred_lstm)
print(f'LSTM Mean Squared Error: {mse_lstm}')
```

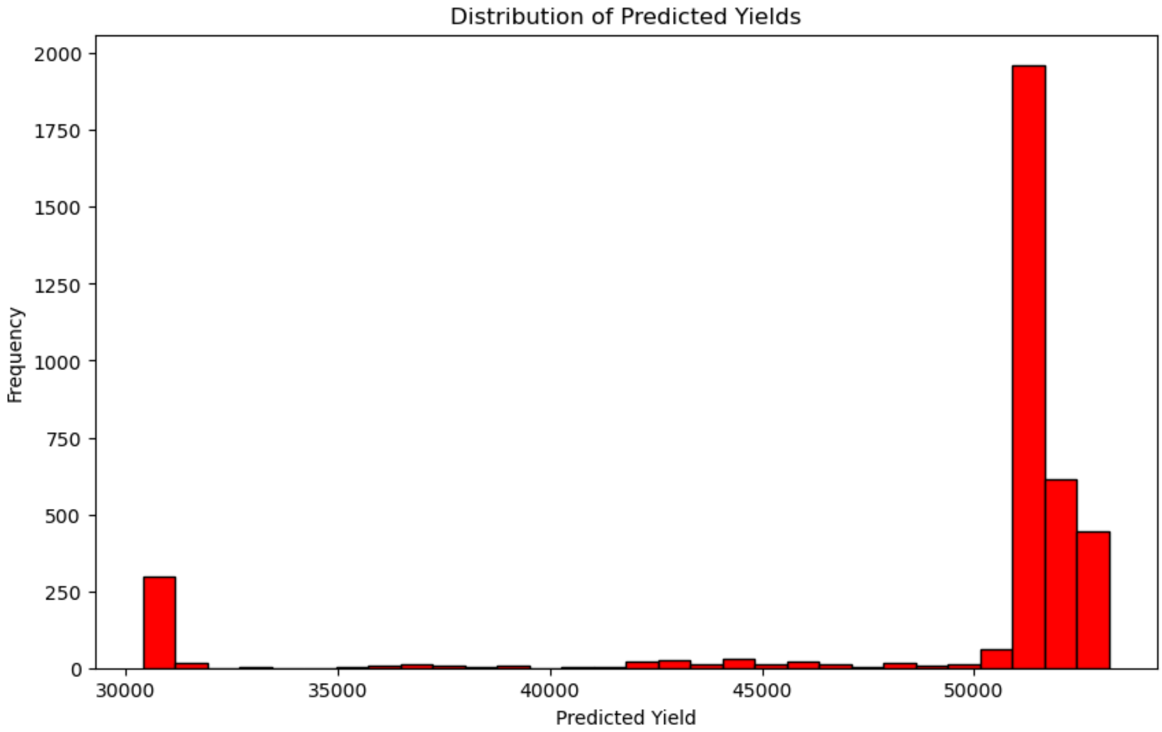
```
114/114 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
LSTM Mean Squared Error: 1.2996516571053496
```

In [169…

```
# visualizing the distribution of predictions
plt.figure(figsize=(10,6))
plt.hist(y_pred_lstm_original, bins=30, color='red', edgecolor='black')
plt.title('Distribution of Predicted Yields')
plt.xlabel('Predicted Yield')
plt.ylabel('Frequency')
```

Out[169…

```
Text(0, 0.5, 'Frequency')
```



In [171…

```
# make the error interpretable
rmse_lstm = np.sqrt(mse_lstm)
print(f'LSTM Root Mean Squared Error: {rmse_lstm}')

normalized_rmse = rmse_lstm / np.mean(y_test_original)
print(f'Normalized RMSE: {normalized_rmse}')
```

```
LSTM Root Mean Squared Error: 1.1400226564000162
Normalized RMSE: 1.414261197996232e-05
```

In [173…

```
# look for NaN values in the predictions
print(f"NaNs in y_pred_lstm_original: {np.isnan(y_pred_lstm_original).sum
print(f"Infs in y_pred_lstm_original: {np.isinf(y_pred_lstm_original).sum
```

```
NaNs in y_pred_lstm_original: 0
Infs in y_pred_lstm_original: 0
```

In [175…

```
# for debugging
print(y_pred_lstm_original.shape)
print(y_test_original.shape)
```

```
(3622, 1)
(3622,)
```

Calculating Risk Score

Just having the predictions is not enough. We also need to analyze the risk.

- We developed a program that first sets low and high thresholds for the various features (like rainfall, temperature, and pesticide usage), as well as the yield.
- We then compare the inputs fed to the model as well as its prediction with these thresholds, and assign each factor a risk weight.

Note: The thresholds were calculated using common techniques and estimations. These can be made more robust by assigning more weight to

recent data, or by seperately training ML models. But this is outside the scope of our project.

In [177…

```
# reverse the log transformation on the target variable
if (yield_df_target_exp == False):
  yield_df[target] = np.expm1(yield_df[target])
  yield_df_target_exp = True
```

Calculating thresholds based on historical data

- As more data points are added, these thresholds will dynamically adjust.

In [179…

```
# calculate thresholds for rainfall based on historic data
rainfall_low_threshold = yield_df['average_rain_fall_mm_per_year'].quanti
rainfall_high_threshold = yield_df['average_rain_fall_mm_per_year'].quant

# calculate thresholds for temperature anomalies
temp_mean = yield_df['avg_temp'].mean()
temp_std = yield_df['avg_temp'].std()
temp_low_threshold = temp_mean - 2 * temp_std  # lower bound for anomalie
temp_high_threshold = temp_mean + 2 * temp_std  # upper bound for anomali

# calculate pesticide usage threshold (90th percentile for high risk)
pesticide_high_threshold = yield_df['pesticides_tonnes'].quantile(0.9)

# calculate predicted yield thresholds
yield_df['historical_yield'] = yield_df.groupby(['Area', 'Item'], observe
predicted_yield_low_threshold = yield_df['historical_yield'] * 0.8  # poo
predicted_yield_high_threshold = yield_df['historical_yield'] * 1.2  # go

# Print calculated thresholds
print(f"Drought Risk Threshold: Rainfall < {rainfall_low_threshold} mm")
print(f"Flood Risk Threshold: Rainfall > {rainfall_high_threshold} mm")
print(f"Temperature Anomaly Thresholds: Low < {temp_low_threshold}, High
print(f"Pesticide High Usage Threshold: > {pesticide_high_threshold}")
```

```
Drought Risk Threshold: Rainfall < 396.200000000003 mm
Flood Risk Threshold: Rainfall > 2041.0 mm
Temperature Anomaly Thresholds: Low < 8.042892737629112, High > 33.7331416
7097304
Pesticide High Usage Threshold: > 63829.77
```

Using these thresholds, we now calculate the overall risk score using the function `calculate_risk_score()`. Here is how it works:

- Using helper functions, we first calculate the risk for each factor (a value between 0 and 1)
- Next, based on the severity of the individual risk scores, penalties are applied. Higher risk scores are penalized more.
- Next, if there are multiple factors with high risk, a small but impactful penalty is added.
- The risk score is normalized to ensure that the output is between 0 and 1.

Note: The penalties can be trained using machine learning models. However, it did not work well with the dataset we had. Our risk function is a working

prototype, and can be improved by using machine learning techniques or better estimation.

In [181…

```
def calculate_risk_score(row, rainfall_low_threshold, rainfall_high_thres

    # risk due to rainfall
    def calculate_rainfall_risk(rainfall):
        if rainfall < rainfall_low_threshold:
            # drought risk
            return min(1.0, (rainfall_low_threshold - rainfall) / rainfal
        elif rainfall > rainfall_high_threshold:
            # flood risk
            return min(1.0, (rainfall - rainfall_high_threshold) / rainfa
        else:
            # calculating how close the value is to the extremes
            distance_from_low = abs(rainfall - rainfall_low_threshold) /
            distance_from_high = abs(rainfall - rainfall_high_threshold)
            return max(0, min(0.3, 1 - (distance_from_low + distance_from

    # risk due to temperature
    def calculate_temperature_risk(temp):
        if temp < temp_low_threshold:
            # cold stress
            return min(1.0, (temp_low_threshold - temp) / temp_low_thresh
        elif temp > temp_high_threshold:
            # heat stress
            return min(1.0, (temp - temp_high_threshold) / temp_high_thre
        else:
            # calculating how close the value is to the extremes
            distance_from_low = abs(temp - temp_low_threshold) / temp_low
            distance_from_high = abs(temp - temp_high_threshold) / temp_h
            return max(0, min(0.3, 1 - (distance_from_low + distance_from

    # risk due to overuse of pesticides
    def calculate_pesticide_risk(pesticides):
        if pesticides > (pesticide_high_threshold * 1.5):
            return 1.0  # extremely high usage
        elif pesticides > pesticide_high_threshold:
            return 0.7  # high usage
        else:
            # calculating how close the value is to the extremes
            distance_from_high = abs(pesticides - pesticide_high_threshol
            return max(0, min(0.3, 1 - distance_from_high))

    # risk due to low yieldsd
    def calculate_yield_risk(historical_yield, predicted_yield):
        # calculate difference between predicte yield and the mean of pas
        yield_deviation = abs(predicted_yield - historical_yield) / histo

        if predicted_yield < (historical_yield * 0.6):
            # extremely below average
            return min(1.0, yield_deviation * 2)
        elif predicted_yield < (historical_yield * 0.8):
            # below average
            return min(0.7, yield_deviation)
        elif predicted_yield > (historical_yield * 1.4):
            # extremely above average
            return min(1.0, yield_deviation)
        else:
```

```
            # within average
            return max(0, min(0.3, yield_deviation))

    # calculate individual risk components
    rainfall_risk = calculate_rainfall_risk(row['average_rain_fall_mm_per
    temperature_risk = calculate_temperature_risk(row['avg_temp'])
    pesticide_risk = calculate_pesticide_risk(row['pesticides_tonnes'])
    yield_risk = calculate_yield_risk(row['historical_yield'], row['predi

    # dynamically adjust weights based on risk severity
    # if a risk is too high (> 0.7), it gets more weight
    # this attempts to reduce the number of false positives
    def dynamic_weighting(risks):
        # more weight to higher risks
        weights = [
            1.5 if risk > 0.7 else
            1.2 if risk > 0.5 else
            1.0 if risk > 0.3 else
            0.8 for risk in risks
        ]
        return weights

    # compute dynamic weights
    weights = dynamic_weighting([
        rainfall_risk,
        temperature_risk,
        pesticide_risk,
        yield_risk
    ])

    # weighted risk calculation with interaction terms
    # normalizing the risk so that the output is between 0 and 1
    risk_score = sum([
        weights[0] * rainfall_risk,
        weights[1] * temperature_risk,
        weights[2] * pesticide_risk,
        weights[3] * yield_risk
    ]) / sum(weights)

    # add interaction penalty for concurrent risks
    # if multiple risks are high, add a small but impactful penalty
    interaction_penalty = sum([
        0.2 if r > 0.5 else 0
        for r in [rainfall_risk, temperature_risk, pesticide_risk, yield_
    ]) * 0.1

    # final risk score
    final_risk = min(1.0, risk_score + interaction_penalty)

    return final_risk
```

In [183…

```
# for debugging
print(len(test_indices))
len((y_pred_lstm))
```

```
3622
```

Out[183… 3622

Map predictions to original data

The predictions and the testing set only use a subset of the features. However, to calculate the risk score, we need all the original features (which may be missing).

Therefore, we map the testing dataset back to the original dataset (`yield_df`). We make use of the `original_indices` array (and the `test_indices` array created during train-test-split) that we created above.

In [185…

```
# create a dataframe to map predictions back to original data
prediction_map = pd.DataFrame({
    'original_index': test_indices,
    'predicted_yield': y_pred_lstm_original.flatten()
})
```

In [187…

```
# merge with original yield_df for calculate_risk_score() function
prediction_results = yield_df.loc[prediction_map['original_index']].copy(
prediction_results['predicted_yield'] = prediction_map['predicted_yield']
# prediction_results['predicted_yield_original'] = np.expm1(prediction_re

# calculate historical yield for reference
prediction_results['historical_yield'] = prediction_results.groupby(['Are
# prediction_results['historical_yield_original'] = np.expm1(prediction_r
```

In [189…

```
# apply calculate_risk_score() function
prediction_results['risk_score'] = prediction_results.apply(
    lambda row: calculate_risk_score(row,
                                     rainfall_low_threshold,
                                     rainfall_high_threshold,
                                     temp_low_threshold,
                                     temp_high_threshold,
                                     pesticide_high_threshold
    ), axis = 1
)
```

In [191…

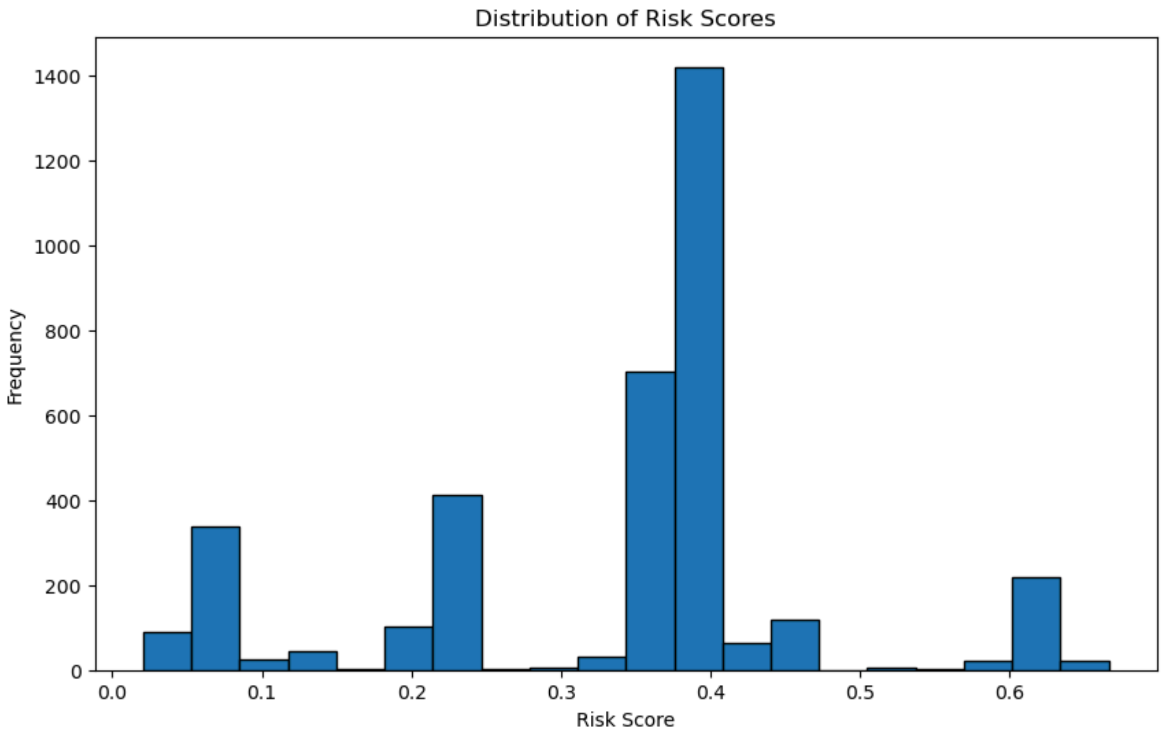
```
# display results
# only the top 5 results are shown here
print(prediction_results[['Area', 'Item', 'Year', 'predicted_yield', 'his
```

```
          Area       Item  Year  predicted_yield  historical_yield  risk_sc
ore
95     Albania      Maize  2013     52297.214844      32739.831239    0.222
539
16     Albania   Soybeans  1992     50714.906250       9776.441233    0.405
004
69     Albania   Soybeans  2006     51523.734375       9776.441233    0.407
648
455  Argentina      Wheat  1994     51523.734375      21514.730769    0.466
154
10     Albania   Soybeans  1991     52428.898438       9776.441233    0.405
004
```

In [193…

```
# visualize risk distribution
plt.figure(figsize=(10,6))
plt.hist(prediction_results['risk_score'], bins=20, edgecolor='black')
plt.title('Distribution of Risk Scores')
```

```
plt.xlabel('Risk Score')
plt.ylabel('Frequency')
plt.show()
```



In [195…

```
# Calculate the difference between predicted_yield and historical_yield
# not a representative result, since historical_yield is the mean of past
# the testing and training loss is more representative

prediction_results['yield_difference'] = prediction_results['predicted_yi

# Calculate the average difference
avg_difference = prediction_results['yield_difference'].mean()

# Print the average difference
print(avg_difference)
```

```
6032.0361204099945
```