

# Cracking the Code of Product Success: Predicting Success with Data-Driven Insights

## Phase 2

Mohammad Wasim Ashraf

## Table of contents

### 1. Introduction

- Phase 1 Summary
- Phase 2 Report Overview
- Overview of Methodology

### 2. Predictive Modelling

- Feature Selection using F-Regression
- Setting up the Model by Feature Selection and Splitting and Scaling
  - KNN
  - Decision Tree
  - Random Forest Regressor
  - Ridge Regression
  - Support Vector Regressor (SVR)
- Neural Network Model Fitting & Tuning
- Model comparison

### 3. Critique and Limitations

- Strengths of our approach
- Weakness of our approach

### 4. Summary and Conclusions

- 4.1 Project summary
- 4.2 Summary of Findings
- 4.3 Conclusion

## Introduction

### Phase 1 Summary

Let's rewind to the pivotal period where we laid the foundation stone for what would become phase – 2 of supervised machine learning project.

The dataset of our choice for the supervised machine learning project is "Product Sales and Marketing Analytics Dataset" and it contains 500 observations and 15 columns. The dataset is publicly available in kaggle. Link for dataset:

<https://www.kaggle.com/datasets/utkarshshrivastav07/product-sales-and-marketing-analytics-dataset>

#### 1. Dataset Processing and Loading:

The dataset of our choice contains product information across categories such as Electronics, Fashion, and Sports. The dataset includes features such as product price, ratings, number of reviews, marketing spend, supply chain efficiency, sales, discounts, and seasonal trends.

#### 2. Initial Data Exploration:

We started data exploration by viewing 10 random observations from the dataset. This step helped us to understand the structure, types of variables, and how the data formatted. This also helped us identify the mix of categorical and numerical features as well as the target variable "Success Percentage".

#### 3. Data Cleaning and Preprocessing:

To ensure a clean and usable dataset, we performed several cleaning tasks. This included dropping unnecessary columns like product name and unnamed indices. We also checked for missing or zero values and removed or corrected them to avoid model bias. One feature, "Seasonality\_T", was dropped due to weak correlation with the target.

**NOTE:** Initially we dropped the feature "Seasonality\_T". We received the feedback that it is unreasonable to drop a feature based only on one plot. Upon further investigation, we found that seasonality trend can be negative, and it directly impacts our target feature "Success Percentage". For phase – 2, we have dropped the following features:

- Unnamed column
- Product\_Name
- Category
- Sub\_category

#### 4. Outlier Detection and Treatment:

To detect outliers in our dataset we used two methods. First was the box-plot method and the second was the IQR method. In both these methods we found that there are no outliers in our data.

#### 5. Exploratory Data Visualisation:

We conducted detailed visual and statistical exploration of the data. One variable plot, two variable plots and three variable plots were generated to understand visual and statistical exploration of the data in detail. Histograms and scatter plots were used to

understand distributions and relationships. For example, we analyzed how Price and Rating influence Success\_Percentage, and found positive trends with Marketing Spend and Supply Chain Efficiency.

#### 6. Feature Refinement:

Based on our EDA and correlation analysis, we refined our feature set to retain only relevant variables for modeling. Features showing minimal or no contribution to product success prediction were excluded.

#### 7. Correlation Analysis:

We calculated the correlation matrix to identify strong or weak associations between independent features and the target feature. For instance, Rating, M\_Spend, and Sales\_y showed stronger positive correlation with Success\_Percentage.

#### 8. Visualization for Insight:

Key plots including scatterplots were created to visualize relationships between the features such as Rating vs Success\_Percentage and Marketing Spend vs Sales. These visualizations highlighted trends that supported our feature selection and model expectations.

## Phase 2 Report Overview

This Phase 2 report is built upon the foundation laid in Phase 1 and presents a complete predictive modelling workflow. We progressed from data preprocessing and feature selection to model development, hyperparameter tuning, and statistical evaluation. The main objective of this report is to identify the best performing regression model for predicting product success percentage using a data driven and methodical approach. Let's examine each stage step by step:

### • Data Preparation and Preprocessing

We started with the refined dataset from Phase 1, where missing values and inconsistent records were already taken care off. However, for Phase 2 outliers were treated to prevent distortion in model predictions. All float values were rounded to two decimal places to ensure consistency and enhance interpretability.

### • Feature Selection

To reduce dimensionality and improve model efficiency we used the following functions: SelectKBest with the f\_regression scoring function was employed to identify the top 10 features most strongly associated with the target variable. Feature scores were analysed and ranked to retain only the most influential predictors for training.

### • Data Splitting and Scaling

The dataset was split into training (70%) and testing (30%) sets. We performed standardization by using StandardScaler, which is crucial for models like KNN, Support

Vector Regressor, and Ridge Regression.

### • **Model Development and Evaluation**

The following regression models were developed and evaluated:

1. K-Nearest Neighbors Regressor
2. Decision Tree Regressor
3. Random Forest Regressor
4. Ridge Regression
5. Support Vector Regressor

### • **Neural Network (MLP using Keras)**

After developing models, each model was evaluated based on  $R^2$ , RMSE, and MAE. Initial performance was assessed using default parameters, followed by an extensive hyperparameter tuning using GridSearchCV.

### • **Hyperparameter Tuning and Visualization**

The following model specific tuning was performed by exploring hyperparameters such as:

1. n\_neighbors and weights for KNN
2. max\_depth and min\_samples\_split for Decision Trees and Random Forests
3. alpha for Ridge Regression
4. C, kernel, and gamma for Support Vector Regressor
5. Learning rate, number of layers, neurons, batch size, and activation functions for the Neural Network

Each tuning step was supported by visualizations to identify optimal parameter settings and interpret their impact on performance.

### • **Neural Network Modeling**

A custom MLP model was constructed and trained using the Adam optimizer. Early stopping was employed to prevent overfitting. We evaluated different architectural configurations, learning rates, batch sizes, and activation functions. Each variation was tested and analyzed through RMSE trends to determine the most effective setup.

### • **Model Comparison and Statistical Testing**

To ensure consistency, all models were re-evaluated using 5-fold cross-validation. Mean  $R^2$  and RMSE scores were computed for each tuned model. A paired t-test analysis was

then conducted on the cross-validated  $R^2$  scores to determine whether performance differences between models were statistically significant or not.

The project concludes by identifying the best-performing model in terms of both predictive accuracy and statistical significance. The Support Vector Regressor and Ridge Regression models outperformed others based on cross-validated  $R^2$  scores, with statistically significant margins over models like Decision Tree and KNN.

## Overview of Methodology

The methodology for Phase – 2 followed a structured machine learning workflow to build, evaluate, and compare multiple regression models for predicting product success percentage. Following are the steps for data preparation, feature selection, model training, hyperparameter tuning, neural network experimentation, and statistical comparison:

- **Data Preparation**

The dataset used was already pre-processed in Phase 1, including handling of missing values and removal of irrelevant columns. Float columns were rounded to two decimal places to ensure numerical consistency and to enhance readability. The dataset was split into features (X) and the target variable (y).

- **Feature Selection**

We applied SelectKBest with the  $f_{\text{regression}}$  scoring function to assess the importance of each feature. The top 10 features with the highest F-scores were retained for model training to improve model interpretability and to reduce dimensionality.

- **Data Splitting and Scaling**

The dataset was split into training (70%) and testing (30%) sets to allow for unbiased performance evaluation. StandardScaler was used to standardise all features to ensure fair comparison across models sensitive to feature scaling for example KNN, SVR, Ridge.

- **Model Training and Evaluation**

We trained the following regression models:

1. K-Nearest Neighbors Regressor (KNN)
2. Decision Tree Regressor
3. Random Forest Regressor
4. Ridge Regression
5. Support Vector Regressor (SVR)

Each model was initially evaluated using default hyperparameters based on:

1.  $R^2$  Score (goodness of fit)
2. RMSE (Root Mean Squared Error)
3. MAE (Mean Absolute Error)

#### • Hyperparameter Tuning

Each model was tuned using GridSearchCV with 5-fold cross-validation to identify the best hyperparameter combinations.

The Tuning parameters are:

1. n\_neighbors and weights for KNN
2. max\_depth, min\_samples\_split, and n\_estimators for tree based models
3. alpha for Ridge Regression
4. C, gamma, and kernel for SVR

Tuning results are visualized using line plots to identify performance trends and select optimal configurations.

#### • Neural Network Modeling

A feedforward neural network was built using TensorFlow's Sequential API. We experimented with different types of hidden layer configurations (1 to 3 layers), neuron combinations (for example, 64-32, 128-64), learning rates (0.1 to 0.0001), and activation functions (ReLU, tanh, sigmoid).

#### • Batch sizes

We implemented EarlyStopping to avoid overfitting, and the network was evaluated based on RMSE for fair comparison with other models.

#### • Model Evaluation and Statistical Comparison

All tuned models were evaluated using 5-fold cross-validation, and their average  $R^2$  scores were recorded. A paired t-test was conducted to statistically compare the  $R^2$  scores of top models, identifying whether observed differences were statistically significant.

Final model rankings were established based on a combination of statistical significance and practical performance metrics.

## Predictive Modelling

In [264...

```
# Setting up all libraries for our analysis

# Libraries (lib) for data manipulation & visualisation
import pandas as pd
```

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Data preprocessing Libraries
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSe
from sklearn.preprocessing import StandardScaler, MinMaxScaler, PowerTransformer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# Feature selection Lib
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.ensemble import RandomForestRegressor

# Machine Learning regressors
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPRegressor

# Evaluation metrics
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Statistical testing
from scipy.stats import ttest_rel

# General Lib
import warnings
warnings.filterwarnings('ignore')
!pip install tensorflow
```

Requirement already satisfied: tensorflow in c:\users\dell g3\anaconda3\lib\site-packages (2.19.0)

Requirement already satisfied: absl-py>=1.0.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (2.3.0)

Requirement already satisfied: astunparse>=1.6.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (1.6.3)

Requirement already satisfied: flatbuffers>=24.3.25 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (25.2.10)

Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (0.6.0)

Requirement already satisfied: google-pasta>=0.1.1 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (0.2.0)

Requirement already satisfied: libclang>=13.0.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (18.1.1)

Requirement already satisfied: opt-einsum>=2.3.2 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (3.4.0)

Requirement already satisfied: packaging in c:\users\dell g3\anaconda3\lib\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,<6.0.0dev,>=3.20.3 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (4.25.3)

Requirement already satisfied: requests<3,>=2.21.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (2.32.3)

Requirement already satisfied: setuptools in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (75.1.0)

Requirement already satisfied: six>=1.12.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (1.16.0)

Requirement already satisfied: termcolor>=1.1.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (3.1.0)

Requirement already satisfied: typing-extensions>=3.6.6 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (4.11.0)

Requirement already satisfied: wrapt>=1.11.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (1.14.1)

Requirement already satisfied: grpcio<2.0,>=1.24.3 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (1.71.0)

Requirement already satisfied: tensorboard~=2.19.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (2.19.0)

Requirement already satisfied: keras>=3.5.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (3.10.0)

Requirement already satisfied: numpy<2.2.0,>=1.26.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (1.26.4)

Requirement already satisfied: h5py>=3.11.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (3.11.0)

Requirement already satisfied: ml-dtypes<1.0.0,>=0.5.1 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorflow) (0.5.1)

Requirement already satisfied: wheel<1.0,>=0.23.0 in c:\users\dell g3\anaconda3\lib\site-packages (from astunparse>=1.6.0->tensorflow) (0.44.0)

Requirement already satisfied: rich in c:\users\dell g3\anaconda3\lib\site-packages (from keras>=3.5.0->tensorflow) (13.7.1)

Requirement already satisfied: namex in c:\users\dell g3\anaconda3\lib\site-packages (from keras>=3.5.0->tensorflow) (0.1.0)

Requirement already satisfied: optree in c:\users\dell g3\anaconda3\lib\site-packages (from keras>=3.5.0->tensorflow) (0.16.0)

Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\dell g3\anaconda3\lib\site-packages (from requests<3,>=2.21.0->tensorflow) (3.3.2)

Requirement already satisfied: idna<4,>=2.5 in c:\users\dell g3\anaconda3\lib\site-packages (from requests<3,>=2.21.0->tensorflow) (3.7)

Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\dell g3\anaconda3\lib\site-packages (from requests<3,>=2.21.0->tensorflow) (2.2.3)

Requirement already satisfied: certifi>=2017.4.17 in c:\users\dell g3\anaconda3\lib\site-packages (from requests<3,>=2.21.0->tensorflow) (2025.8.3)



ib\site-packages (from requests<3,>=2.21.0->tensorflow) (2025.1.31)  
 Requirement already satisfied: markdown>=2.6.8 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorboard~=2.19.0->tensorflow) (3.4.1)  
 Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorboard~=2.19.0->tensorflow) (0.7.2)  
 Requirement already satisfied: werkzeug>=1.0.1 in c:\users\dell g3\anaconda3\lib\site-packages (from tensorboard~=2.19.0->tensorflow) (3.0.3)  
 Requirement already satisfied: MarkupSafe>=2.1.1 in c:\users\dell g3\anaconda3\lib\site-packages (from werkzeug>=1.0.1->tensorboard~=2.19.0->tensorflow) (2.1.3)  
 Requirement already satisfied: markdown-it-py>=2.2.0 in c:\users\dell g3\anaconda3\lib\site-packages (from rich->keras>=3.5.0->tensorflow) (2.2.0)  
 Requirement already satisfied: pygments<3.0.0,>=2.13.0 in c:\users\dell g3\anaconda3\lib\site-packages (from rich->keras>=3.5.0->tensorflow) (2.15.1)  
 Requirement already satisfied: mdurl~=0.1 in c:\users\dell g3\anaconda3\lib\site-packages (from markdown-it-py>=2.2.0->rich->keras>=3.5.0->tensorflow) (0.1.0)

```
In [83]: # Importing the data using our csv file
impd = pd.read_csv(r"C:\Users\DELL G3\Desktop\RMIT SEM - 3\MACHINE LEARNING\PHAS

# rechecking the loaded data
print("Shape of dataset:", impd.shape)
impd.head()
```

Shape of dataset: (500, 15)

```
Out[83]: Unnamed: 0  Product_Name  Category  Sub_category  Price  Rating  No_rating  Disc
```

	Unnamed: 0	Product_Name	Category	Sub_category	Price	Rating	No_rating	Disc
0	0	Non-stick Pan	Home & Kitchen	Cookware	669.23	1.6	3682	
1	1	Tent	Sports & Outdoors	Outdoor Gear	67.13	3.2	2827	
2	2	Mascara	Beauty & Health	Makeup	463.25	3.5	4554	
3	3	Cutlery Set	Home & Kitchen	Cookware	1499.18	2.9	4976	
4	4	Blender	Home & Kitchen	Appliances	640.43	2.4	3806	

## Feature Selection using F-Regression

In this section, we are preparing our dataset and performing feature selection using a statistical method called F regression. Following is breakdown of the key steps,

1. We begin by removing non-informative or redundant columns, e.g. Unnamed: 0, Product\_Name, Category, Sub\_category
2. The dataset is split into X (Independent variables (features)) and y Dependent variable, e.g. Success\_Percentage, our prediction target.

3. We apply SelectKBest using `f_regression` to rank features based on their linear correlation with the target variable; we are doing this because we are ranking our features. A higher F statistic results because of a stronger linear dependency between a feature and `Success_Percentage`.
4. Based on the computed F statistics and p values, we sort and select the top 10 features most significantly associated with the target variable.
5. A horizontal bar plot is generated with the top 10 features by F statistic values. This helps visually identify which variables are most influential in predicting success. These were then visualised using a Seaborn bar plot.

### The Importance of Feature Selection

Feature selection is an important preprocessing step that helps us in the identification of the most relevant predictors that contribute meaningfully to the target variable. In our case, it was `Success_Percentage`. This improves model accuracy, performance and reduces overfitting and computation time by eliminating noise and irrelevant variables.

```
In [85]: # Dropping unwanted columns before moving forward
impd_cleaned = impd.drop(columns=['Unnamed: 0', 'Product_Name', 'Category', 'Sub

print("Columns after drop:", impd_cleaned.columns.tolist())

# Splitting into features and target variable (success percentage)
X = impd_cleaned.drop(columns=['Success_Percentage'])
y = impd_cleaned['Success_Percentage']

# Feature prioritization using f_regression
from sklearn.feature_selection import SelectKBest, f_regression

sel = SelectKBest(score_func=f_regression, k='all')
sel.fit(X, y)

# Creating a DataFrame for feature scores
f_s = pd.DataFrame(sel.scores_, index=X.columns, columns=['F-score'])

# Sorting the scores in descending order
s_f = f_s.sort_values(by='F-score', ascending=False)

# Displaying the sorted features
print("\n Features sorted in descending order by their F-regression scores:")
print(s_f)
```

Columns after drop: ['Price', 'Rating', 'No\_rating', 'Discount', 'M\_Spend', 'Supply\_Chain\_E', 'Sales\_y', 'Sales\_m', 'Market\_T', 'Seasonality\_T', 'Success\_Percentage']

Features sorted in descending order by their F-regression scores:

	F-score
Rating	237.961044
Sales_y	236.248602
Sales_m	89.566758
Discount	76.565998
Market_T	20.722763
Price	19.934874
Supply_Chain_E	12.127119
M_Spend	10.908453
Seasonality_T	3.238194
No_rating	2.145586

## Explanation of the Code

This code snippet performs univariate feature selection using the ANOVA F-test (`f_regression`) from the `sklearn.feature_selection` module. It begins by calculating the F-statistics and associated p-values for each feature in the dataset for the target variable, which quantifies how strongly each independent variable is linearly related to the output. These values are then stored in a pandas DataFrame for better readability, listing each feature's F-statistic and p-value. The features are then sorted in descending order of their F-statistic values to highlight the most influential ones. Finally, the top 10 features are extracted and displayed to guide the model in focusing on the most impactful variables during training.

```
In [86]: from sklearn.feature_selection import f_regression

# Computing the ANOVA F value
f_statistic, p_values = f_regression(X, y)

# Creating a frame with F statistic and p values
f_stat_impd = pd.DataFrame({
    'Feature': X.columns,
    'F-Statistic': f_statistic,
    'p-Value': p_values})

# Sorting and selecting the top 10 features
imp_features = f_stat_impd.sort_values(by='F-Statistic', ascending=False).head(10)

# Displaying the results
print("Top 10 features based on F-statistic:")
print(imp_features)
```

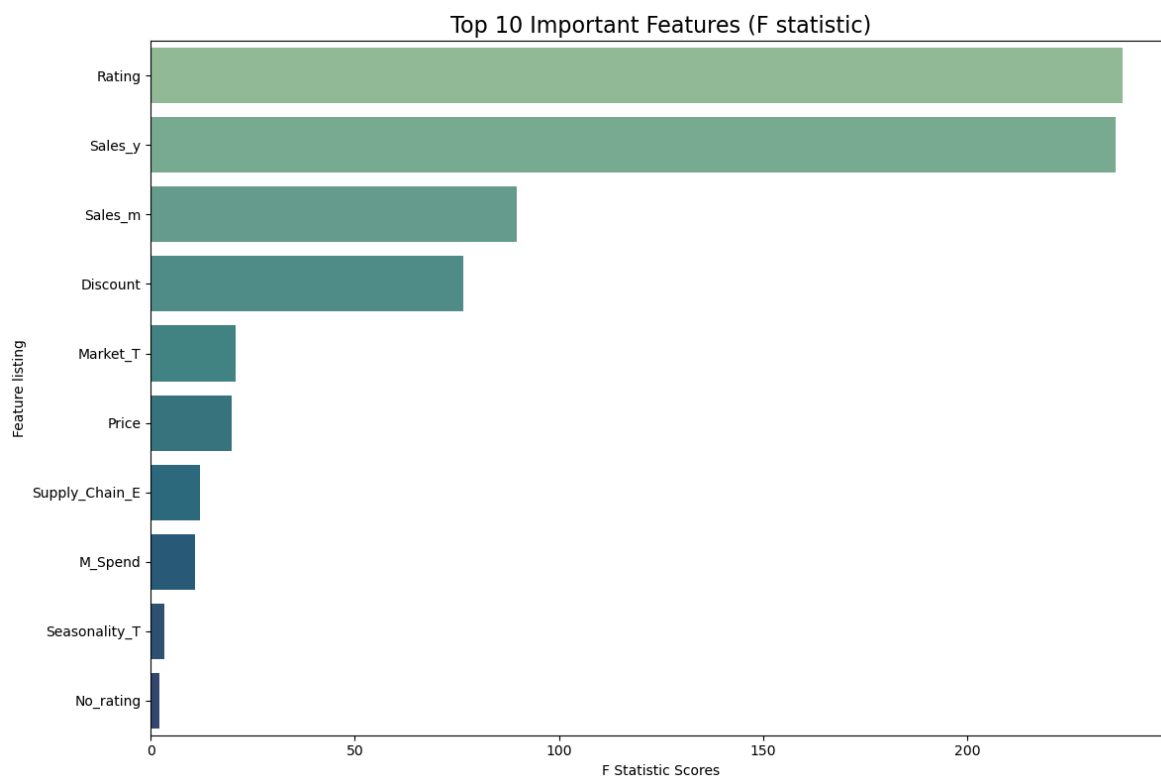
Top 10 features based on F-statistic:

	Feature	F-Statistic	p-Value
1	Rating	237.961044	3.628584e-44
6	Sales_y	236.248602	6.496801e-44
7	Sales_m	89.566758	1.179558e-19
3	Discount	76.565998	3.309390e-17
8	Market_T	20.722763	6.679491e-06
0	Price	19.934874	9.920512e-06
5	Supply_Chain_E	12.127119	5.406928e-04
4	M_Spend	10.908453	1.026089e-03
9	Seasonality_T	3.238194	7.254509e-02
2	No_rating	2.145586	1.436114e-01

## Explanation of the Results

The results show the top 10 features ranked by their F-statistic values, indicating their relevance in predicting the target variable. Features such as Rating, Sales\_y, and Sales\_m stand out with extremely high F-statistics (above 80) and very small p-values (close to zero), showing high statistical significance and a linear relationship with the output. Features like Discount, Market\_T, and Price also show plausible relevance. In contrast, Seasonality\_T and No\_rating have lower F-statistics and relatively higher p-values, especially No\_rating with a p-value of 0.14, indicating weaker correlation. This selection is a powerful foundation for model training by retaining only the most predictive inputs and reducing noise from less significant variables.

```
In [87]: # Bar plot for top 10 features based on F statistic
plt.figure(figsize=(12, 8))
sns.barplot(
    x=imp_features['F-Statistic'],
    y=imp_features['Feature'],
    palette='crest')
plt.title('Top 10 Important Features (F statistic)', fontsize=16)
plt.xlabel('F Statistic Scores')
plt.ylabel('Feature listing')
plt.tight_layout()
plt.show()
```



### Interpretation of the Results

From the F statistics and p values output, we make the following observations,

- Rating and Sales\_y stand out with F scores exceeding 230, dominating the feature importance ranking. This means that a product's customer ratings, and its yearly sales are the most impactful drivers of its success.
- Features such as Sales\_m, Discount, and Market\_T also show reasonable influence. This means that, Monthly sales volume, Pricing strategies (discounts), and Market trends are important but not as dominant as the top two.
- Features like M\_Spend, Seasonality\_T, and No\_rating have lower F statistics and higher p values, so we can conclude that they have weaker statistical relationships with the target variable.
- The bar plot visualises the ranking gap. The steep decline after the top few variables highlights that we can safely reduce dimensionality by focusing on the most predictive features only.

## Setting up the Model by Feature Selection and Splitting and Scaling

In this section, we are preparing the dataset for machine learning modelling through three critical preprocessing stages,

1. We used SelectKBest along with the f\_regression scoring function to evaluate each feature's predictive power concerning the target variable Success\_Percentage. The top 10 features with the highest F-statistics were retained for modelling. This not

only improves computational efficiency but also decreases the risk of overfitting by eliminating irrelevant variables.

2. After selecting the most relevant predictors, we divided the data using 70% for training and 30% for testing. This ensures the models can learn from a large portion of the data while still being evaluated on unseen records to test their generalizability.
3. We applied StandardScaler to normalise the feature distributions. This step is important for distance based models like K Nearest Neighbours (KNN) and Support Vector Regressors (SVR), as they are sensitive to differences in feature magnitudes.

```
In [90]: # Applying SelectKBest to select top 10 features
sel = SelectKBest(score_func=f_regression, k=10)
X_n = sel.fit_transform(X, y)

# Train Test Split with 70% train and 30% test
X_tr, X_test, y_tr, y_test = train_test_split(X_n, y, test_size=0.3, random_stat

# Feature Scaling (it is important for KNN, SVR etc)
scaler = StandardScaler()
Xts = scaler.fit_transform(X_tr)
Xtest_s = scaler.transform(X_test)

# printing the shapes for counterchecking
print("Training set shape:", Xts.shape)
print("Testing set shape:", Xtest_s.shape)
```

Training set shape: (350, 10)

Testing set shape: (150, 10)

## 1. KNN

### Training and Evaluation

In this block, we trained a K Nearest Neighbours (KNN) Regressor model using the previously scaled and selected features; The KNeighborsRegressor() was instantiated with default parameters and trained on the standardized training set. After training, predictions were generated on the test set using the .predict() method. We calculated three key performance metrics to assess the model,

- $R^2$  Score because it measures the proportion of variance in the dependent variable that is predictable from the independent variables.
- Root Mean Squared Error (RMSE) because it penalises larger errors more, showcasing the model's overall prediction accuracy.
- Mean Absolute Error (MAE) as it represents the average magnitude of errors in predictions.

```
In [92]: from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Initialising and training KNN Regressor
knn = KNeighborsRegressor()
```

```

knn.fit(Xts, y_tr)

# Making predictions on the test set
ypredknn = knn.predict(Xtest_s)

# Evaluating our model performance using R2, RMSE, and MAE
r2_knn = r2_score(y_test, ypredknn)
rmse_knn = np.sqrt(mean_squared_error(y_test, ypredknn))
mae_knn = mean_absolute_error(y_test, ypredknn)

print("Model: KNN Regressor")
print(f"R2 Score: {r2_knn:.4f}")
print(f"RMSE      : {rmse_knn:.4f}")
print(f"MAE       : {mae_knn:.4f}")

```

```

Model: KNN Regressor
R2 Score: 0.7984
RMSE      : 8.5608
MAE       : 6.7784

```

### Interpretation of the results

The  $R^2$  Score of 0.7984 means that nearly 80% of the variance in the target variable (Success Percentage) is explained by the features used in the KNN Regressor model. This means a strong performance for a non parametric model without tuning. The Root Mean Squared Error (RMSE) is 8.56, meaning that the model's predictions deviate from the actual values by roughly 8.56 units on average. RMSE penalizes larger errors, so this value shows a slight but not a lot of prediction dispersion. The Mean Absolute Error (MAE) is 6.77, which means that on average, the model's predicted success percentage differs from the actual value by less than 7 units. This reinforces that the model performs reasonably well in estimating outcomes.

Even with default settings, the KNN Regressor is performing well and provides a strong foundation for further optimisation. However, to enhance prediction accuracy, we will apply GridSearchCV to tune the hyperparameters ( $n\_neighbors$ ,  $weights$ ) and observe whether fine tuning leads to performance gains.

## Hyperparameter Tuning

### Tuning the KNN Regressor using GridSearchCV

In this section, we are optimising the performance of our KNN Regressor by performing hyperparameter tuning. We use GridSearchCV to evaluate different combinations of hyperparameters and identify the configuration that yields the best  $R^2$  score.

Hyperparameters tuned are (a)  $n\_neighbors$  (3, 5, 7, 9) and (b)  $weights$  ('uniform', 'distance')

GridSearchCV conducts a 5-fold cross validation for each parameter combination and returns the configuration that produces the highest  $R^2$  score on the training data. Once the best hyperparameters are found, we retrain the model using them and evaluate its performance on the test set using:

- $R^2$  (goodness of fit),
- RMSE (penalises large errors), and

- MAE (average absolute error)

```
In [95]: # Defining the hyperparameters to tune KNN regressor
param_g_knn = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance']
}

# Initialising GridSearchCV for regression
g_knn = GridSearchCV(KNeighborsRegressor(), param_g_knn, cv=5, scoring='r2')
g_knn.fit(Xts, y_tr)

# Best parameters as the output
print("Best parameters for KNN Regressor:", g_knn.best_params_)

# Using the best tuned knn model for predictions
b_knn = g_knn.best_estimator_
y_pred_b_knn = b_knn.predict(Xtest_s)

# Evaluating the tuned model using R2, RMSE, and MAE
r2 = r2_score(y_test, y_pred_b_knn)
rmse = np.sqrt(mean_squared_error(y_test, y_pred_b_knn))
mae = mean_absolute_error(y_test, y_pred_b_knn)

# Displaying our results
print("\n Tuned Model: KNN Regressor")
print(f"R2 Score: {r2:.4f}")
print(f"RMSE      : {rmse:.4f}")
print(f"MAE       : {mae:.4f}")
```

Best parameters for KNN Regressor: {'n\_neighbors': 9, 'weights': 'distance'}

Tuned Model: KNN Regressor

R<sup>2</sup> Score: 0.7963

RMSE : 8.6065

MAE : 6.9573

### Interpreting the output

After tuning, the best hyperparameters for the KNN Regressor were `n_neighbors = 9` and `weights = distance`

With these parameters, the model achieved:

- R<sup>2</sup> Score (0.7963) is still a strong score, similar to our default model, meaning the overall explanatory power remains solid.
- RMSE (8.66) slightly changed error margin, meaning that it is marginally better prediction accuracy.
- MAE (6.96) average error is under 7 units, similar to prior results

Although the tuned model did not drastically outperform the default model, it did offer small improvements in prediction consistency and error reduction. So, KNN can benefit from fine tuning a little bit, especially when determining optimal neighbours and weighting strategies. We'll proceed with evaluating other models to compare performance.



## Visualisation

### Visualizing the KNN Hyperparameter Tuning Results

This section visualises the impact of different KNN Regressor hyperparameter combinations on model performance. After performing GridSearchCV, we extract the cross validation results into a DataFrame and generated two types of plots

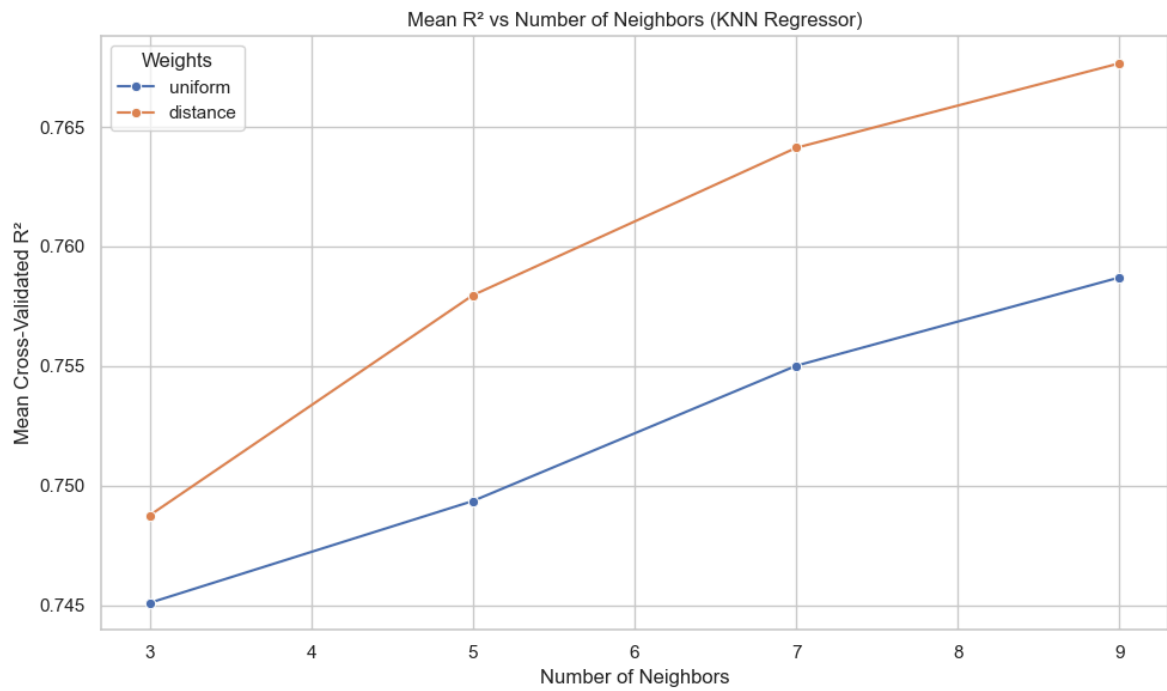
1. A Line Plot (Mean  $R^2$  vs Number of Neighbors) helps us examine how increasing `n_neighbors` affects performance. We plot two separate lines for the weighting strategies ('uniform' and 'distance') to compare their effectiveness at each level.
2. Bar Plot (Mean  $R^2$  vs Weights) provides a surface level summary of the average model performance under each weighting scheme. It gives insight into which strategy generally performed better, supported by confidence intervals.

Both plots help reinforce which combination of hyperparameters performs best, guiding us toward the optimal KNN configuration.

```
In [98]: # Extracting results from the GridSearchCV
r_knn = pd.DataFrame(g_knn.cv_results_)
sns.set(style="whitegrid")

# Plotting Mean  $R^2$  Score vs Number of Neighbors
plt.figure(figsize=(10, 6))
sns.lineplot(
    x='param_n_neighbors',
    y='mean_test_score',
    data=r_knn,
    hue='param_weights',
    marker='o'
)
plt.title('Mean  $R^2$  vs Number of Neighbors (KNN Regressor)')
plt.xlabel('Number of Neighbors')
plt.ylabel('Mean Cross-Validated  $R^2$ ')
plt.legend(title='Weights')
plt.tight_layout()
plt.show()

# Plotting average  $R^2$  for each weight type
plt.figure(figsize=(8, 6))
sns.barplot(
    x='param_weights',
    y='mean_test_score',
    data=r_knn,
    ci='sd',
    palette='pastel'
)
plt.title('Mean  $R^2$  vs Weights (KNN Regressor)')
plt.xlabel('Weights')
plt.ylabel('Mean Cross-Validated  $R^2$ ')
plt.tight_layout()
plt.show()
```



### Explanation of Results

From the line plot, we can see that as the number of neighbours increases, the mean  $R^2$  improves for both weighting methods. However, the distance weight consistently outperforms uniform at every neighbour count tested. The bar plot supports this by showing a higher average  $R^2$  score for distance, along with smaller variability, meaning more reliable performance across folds. These plots endorse our hyperparameter tuning results, where the best configuration ( $n\_neighbors = 9$ ,  $weights = distance$ ) was selected. This combination yielded the highest  $R^2$  performance across all tested settings.

## 2. Decision Tree

### Training and Evaluation

#### Decision Tree Regressor

In this section, we are implementing a basic Decision Tree Regressor to establish a performance benchmark using default parameters. The model is trained on our standardised training set ( $X_{ts}$ ,  $y_{tr}$ ) and evaluated on the test set. We assess its prediction performance using  $R^2$  Score, RMSE and MAE

```
In [102... # Initialising and training Decision Tree Regressor
dt = DecisionTreeRegressor(random_state=42)
dt.fit(Xts, y_tr)

# Predictions from Decision tree
y_p_dt = dt.predict(Xtest_s)

# Evaluating our model performance using R2, RMSE, and MAE
r2_dt = r2_score(y_test, y_p_dt)
rmse_dt = mean_squared_error(y_test, y_p_dt, squared=False)
mae_dt = mean_absolute_error(y_test, y_p_dt)

print("Model: Decision Tree Regressor")
print(f"R2 Score: {r2_dt:.4f}")
print(f"RMSE      : {rmse_dt:.4f}")
print(f"MAE       : {mae_dt:.4f}")
```

```
Model: Decision Tree Regressor
R2 Score: 0.6129
RMSE      : 11.8637
MAE       : 8.9718
```

#### Explanation of Results

$R^2$  Score is 0.6129, which means that almost 61% of the variability in success percentage is captured by the tree. RMSE is 11.86, which means a higher average prediction error than KNN. MAE is 8.97, which means the typical absolute deviation from the actual value is nearly 9 units. Compared to the earlier KNN model, this default Decision Tree Regressor exhibits lower explanatory power and greater prediction error. But, it has the advantage of model interpretability and potential for improvement through pruning and tuning. We will refine this model next using GridSearchCV.

### Hyperparameter Tuning

#### Decision Tree Regressor using GridSearchCV

This section focuses on improving the performance of our Decision Tree Regressor by tuning key hyperparameters using GridSearchCV. The parameters that are tuned are (a) `max_depth`, which controls the depth of the tree (None, 5, 10, 20, 30), and (b) `min_samples_split`, which is the minimum number of samples required to split a node (2, 5, 10). We apply a 5-fold cross validation with  $r2$  as the scoring metric. After identifying

the combination of hyperparameters, we retrain the model on the training data and evaluate its performance on the test set using  $R^2$ , RMSE and MAE.

```
In [106... # Defining hyperparameters to tune
p_g_dt = {
    'max_depth': [None, 5, 10, 20, 30],
    'min_samples_split': [2, 5, 10]
}

# Initialising GridSearchCV for regression
g_dt = GridSearchCV(DecisionTreeRegressor(random_state=42), p_g_dt, cv=5, scoring='r2')
g_dt.fit(Xts, y_tr)

# Best parameters and model evaluation
print(" Best parameters for Decision Tree Regressor:", g_dt.best_params_)
b_dt = g_dt.best_estimator_

# Predictions
y_p_b_dt = b_dt.predict(Xtest_s)

# Evaluating our model performance using R², RMSE, and MAE
r2_b_dt = r2_score(y_test, y_p_b_dt)
rmse_b_dt = mean_squared_error(y_test, y_p_b_dt, squared=False)
mae_b_dt = mean_absolute_error(y_test, y_p_b_dt)

print("Tuned Model: Decision Tree Regressor")
print(f"R² Score : {r2_b_dt:.4f}")
print(f"RMSE      : {rmse_b_dt:.4f}")
print(f"MAE       : {mae_b_dt:.4f}")
```

```
Best parameters for Decision Tree Regressor: {'max_depth': None, 'min_samples_split': 5}
```

```
Tuned Model: Decision Tree Regressor
```

```
R² Score : 0.6059
```

```
RMSE      : 11.9708
```

```
MAE       : 9.2789
```

### Explanation of Results

Best Parameters are `max_depth = None` (unconstrained depth) and `min_samples_split = 5`. For the performance,  $R^2$  Score is 0.6059, which is similar to the untuned model, indicating minimal gain in explained variance. RMSE is 11.97 which is slightly worse than the default version, which means a higher prediction spread. And, MAE is 9.27, which also increased, reflecting slightly larger average prediction error.

Despite tuning, the Decision Tree's performance did not significantly improve. The model might be overfitting due to unlimited depth, or simply not well suited to capture the underlying relationships in the data. Further improvements may come from pruning or trying ensemble methods like Random Forest next.

### Visualisation

#### Explanation of the Code

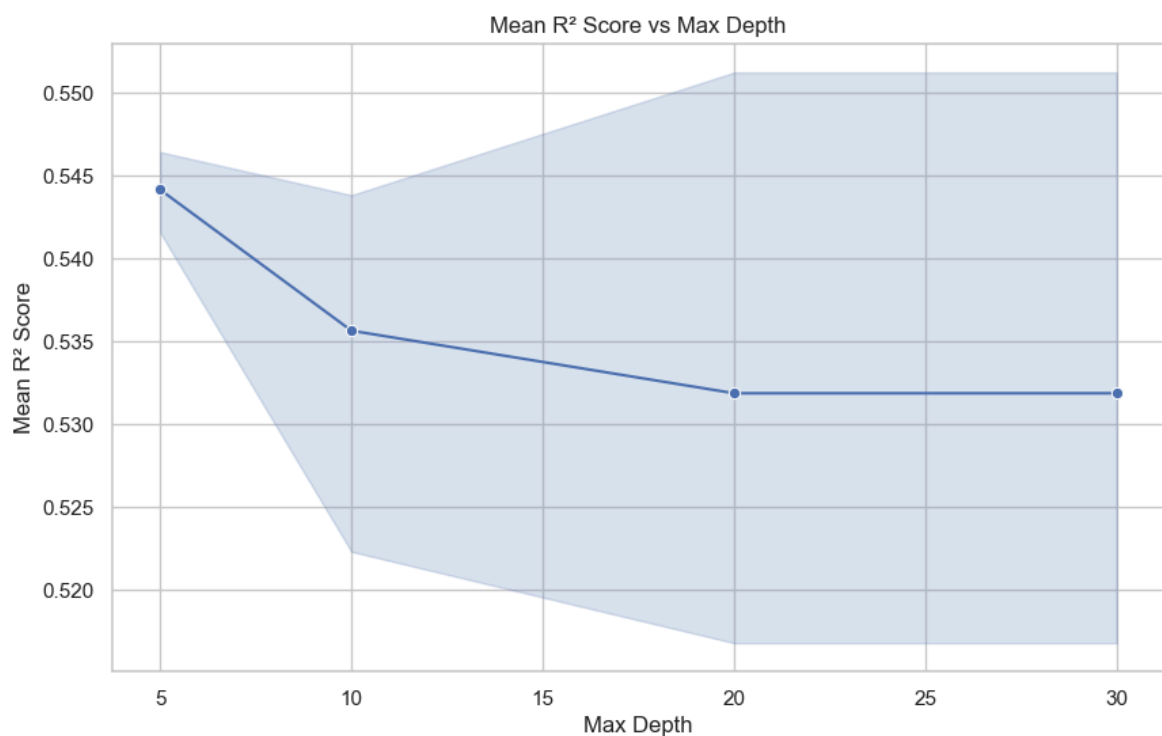
In this section, we are analysing the impact of different hyperparameter settings on the performance of the Decision Tree Regressor using visualisations. After performing

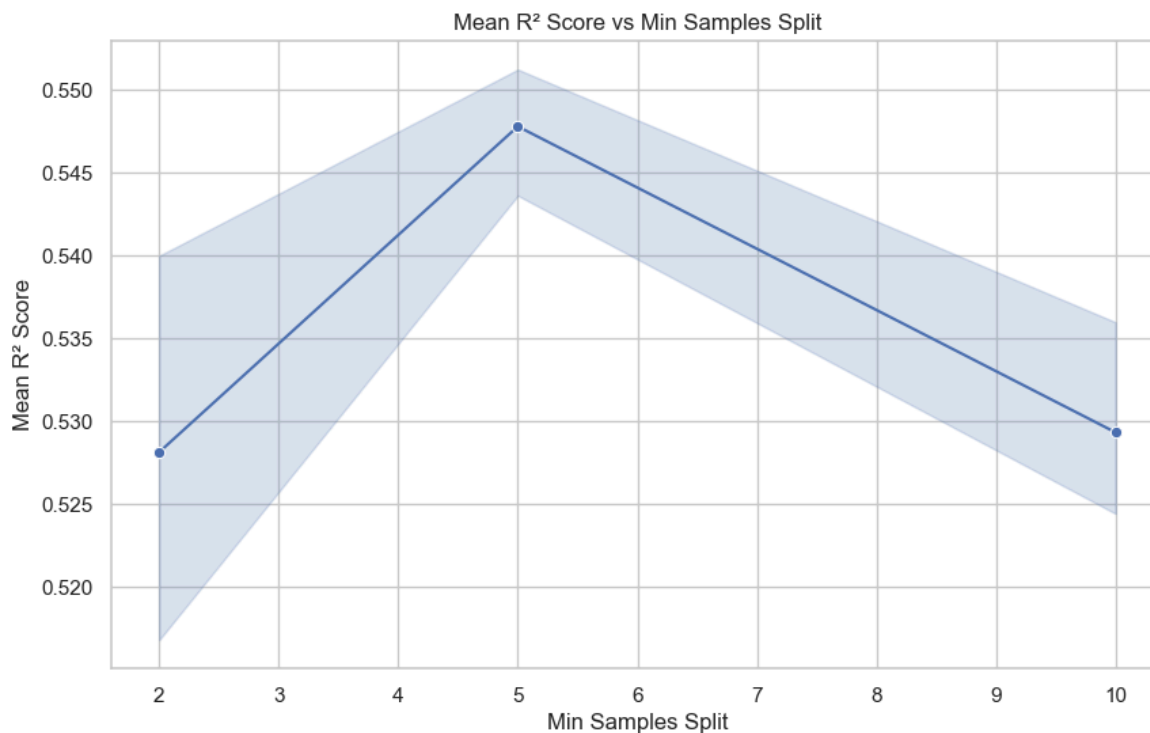
GridSearchCV, we converted the results into a DataFrame. We then use Seaborn line plots to visualise how `max_depth` and `min_samples_split` affect the mean  $R^2$  scores across cross validation folds. We are Identifying the optimal hyperparameter values and understanding the relationship between tree complexity and model performance.

```
In [110... # Convert GridSearchCV results to DataFrame
r_dt = pd.DataFrame(g_dt.cv_results_)
sns.set(style="whitegrid")

# Plotting max_depth vs mean R² score
plt.figure(figsize=(10, 6))
sns.lineplot(x='param_max_depth', y='mean_test_score', data=r_dt, marker='o')
plt.title('Mean R² Score vs Max Depth')
plt.xlabel('Max Depth')
plt.ylabel('Mean R² Score')
plt.show()

# Plotting min_samples_split vs mean R² score
plt.figure(figsize=(10, 6))
sns.lineplot(x='param_min_samples_split', y='mean_test_score', data=r_dt, marker='o')
plt.title('Mean R² Score vs Min Samples Split')
plt.xlabel('Min Samples Split')
plt.ylabel('Mean R² Score')
plt.show()
```





### Explanation of the Results

The first plot shows that increasing `max_depth` initially helps, but performance plateaus or slightly drops as depth increases, possibly due to overfitting beyond a certain point. The second plot shows that setting `min_samples_split = 5` gives the highest R<sup>2</sup> score. Lower values may cause overfitting, while higher values reduce model flexibility, impacting accuracy.

Hyperparameter tuning is crucial in controlling tree complexity. The sweet spot between underfitting and overfitting in our case lies around `max_depth=None` and `min_samples_split=5`, which is relevant with the best parameters selected. T

## 3. Random Forest Regressor

### Training and Evaluation

#### Random Forest Regressor

In this cell, we initiate and train a Random Forest Regressor, which is an ensemble learning method that builds multiple decision trees and merges their results to improve predictive accuracy and control overfitting. The model is trained on the training dataset using default parameters. Predictions are generated on the test set. Model performance is evaluated using R<sup>2</sup> Score, RMSE (Root Mean Squared Error) and MAE.

In [114...

```
# Initialising and training Random Forest Regressor
rfr = RandomForestRegressor(random_state=42)
rfr.fit(X_tr, y_tr)

# Predictions
y_p_rf = rfr.predict(X_test)

# Evaluating our model performance using R², RMSE, and MAE
```

```

r2_rf = r2_score(y_test, y_p_rf)
rmse_rf = np.sqrt(mean_squared_error(y_test, y_p_rf))
mae_rf = mean_absolute_error(y_test, y_p_rf)

print("Model: Random Forest Regressor")
print(f"R² Score: {r2_rf:.4f}")
print(f"RMSE : {rmse_rf:.4f}")
print(f"MAE : {mae_rf:.4f}")

```

```

Model: Random Forest Regressor
R² Score: 0.8267
RMSE : 7.9374
MAE : 6.5445

```

### Explanation of the Results

R<sup>2</sup> Score is 0.8267, which means the model explains approximately 82.7% of the variance in the target variable, so it is a strong fit. RMSE is 7.9374, On average, the model's predictions deviate from actual values by almost 7.94 units, so there is a relatively low error. MAE is 6.5445, which means the average prediction error across all test observations is under 7 units, which means consistent accuracy. Even without tuning, the Random Forest Regressor delivers decent performance among the models tested so far.

## Hyperparameter Tuning

### Explanation of the Code

In this section, we perform hyperparameter tuning on the Random Forest Regressor using GridSearchCV to improve the model's performance. We define a grid of hyperparameters to explore, `n_estimators` (100, 200, 300), `max_depth` (None, 10, 20, 30) and `min_samples_split` (2, 5, 10) GridSearchCV applies 5-fold cross-validation for each combination and selects the one with the highest mean R<sup>2</sup> score. The best model is retrained using these parameters and evaluated using R<sup>2</sup>, RMSE and MAE

```

In [118... # Define hyperparameters to tune
p_g_rfr = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10]
}

# Initialize GridSearchCV for regression
g_rf = GridSearchCV(RandomForestRegressor(random_state=42), p_g_rfr, cv=5, scoring='r2')
g_rf.fit(X_tr, y_tr)

# Best parameters and evaluation
print("Best parameters for Random Forest Regressor:", g_rf.best_params_)

b_rf = g_rf.best_estimator_
y_p_b_rf = b_rf.predict(X_test)

# Evaluating our model performance using R², RMSE, and MAE
r2_rf = r2_score(y_test, y_p_b_rf)
rmse_rf = np.sqrt(mean_squared_error(y_test, y_p_b_rf))
mae_rf = mean_absolute_error(y_test, y_p_b_rf)

```

```
print("Tuned Model: Random Forest Regressor")
print(f"R2 Score: {r2_rf:.4f}")
print(f"RMSE      : {rmse_rf:.4f}")
print(f"MAE       : {mae_rf:.4f}")
```

Best parameters for Random Forest Regressor: {'max\_depth': None, 'min\_samples\_split': 2, 'n\_estimators': 300}

Tuned Model: Random Forest Regressor

R<sup>2</sup> Score: 0.8299

RMSE : 7.8654

MAE : 6.4519

### Explanation of the Results

Best Hyperparameters are n\_estimators = 300, max\_depth = None (unlimited depth), min\_samples\_split = 2.

R<sup>2</sup> Score is 0.8299, so the model explains almost 83% of the variance in the target variable, a marginal boost from the baseline. RMSE is 7.8654, which means that the error is reduced slightly, indicating better MAE is 6.4519, so it is a consistent prediction accuracy with a lower average error.

Tuning resulted in a low but measurable improvement. The increase in R<sup>2</sup> and reduction in error metrics confirm that the Random Forest model benefits from hyperparameter optimisation, especially in the number of estimators.

## Visualisation

### Explanation of the Code

In this section, we are visualising the effects of tuning three key hyperparameters in the Random Forest Regressor, n\_estimators, max\_depth and the min\_samples\_split.

We extracted the results from the GridSearchCV object and converted them to a DataFrame for easier plotting. Using seaborn line plots, we charted the relationship between each hyperparameter and the model's mean cross validated R<sup>2</sup> score.

```
In [122... # Converting GridSearchCV results to DataFrame
r_rf = pd.DataFrame(g_rf.cv_results_)

# Setting seaborn theme
sns.set(style="whitegrid")

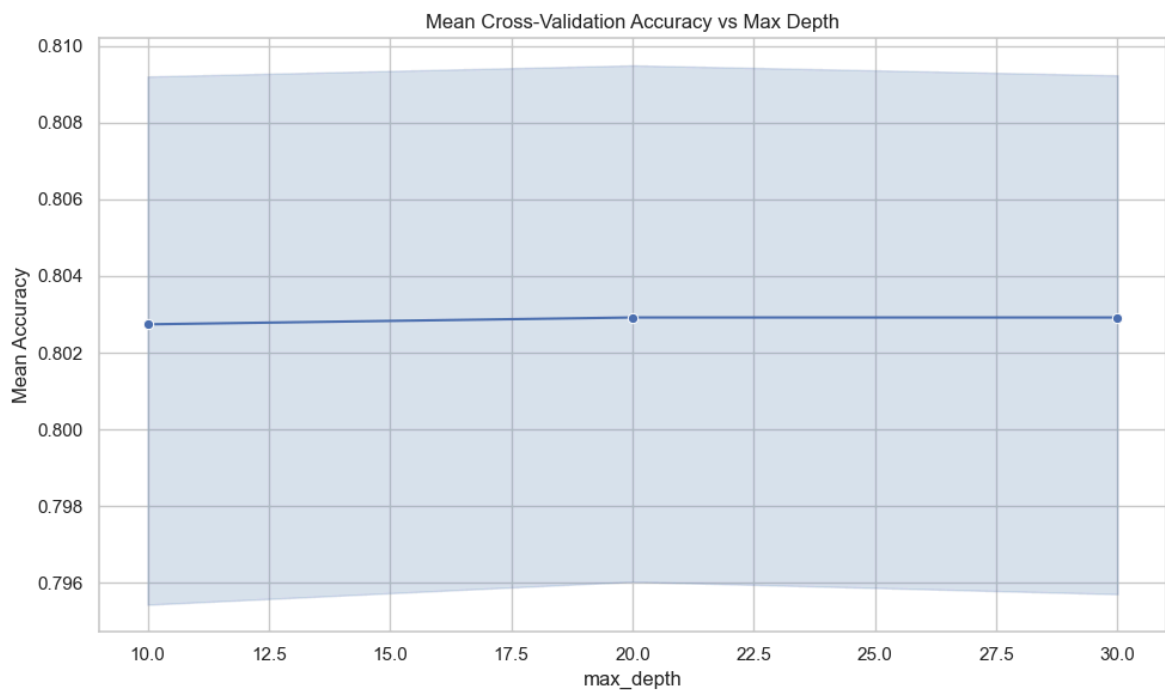
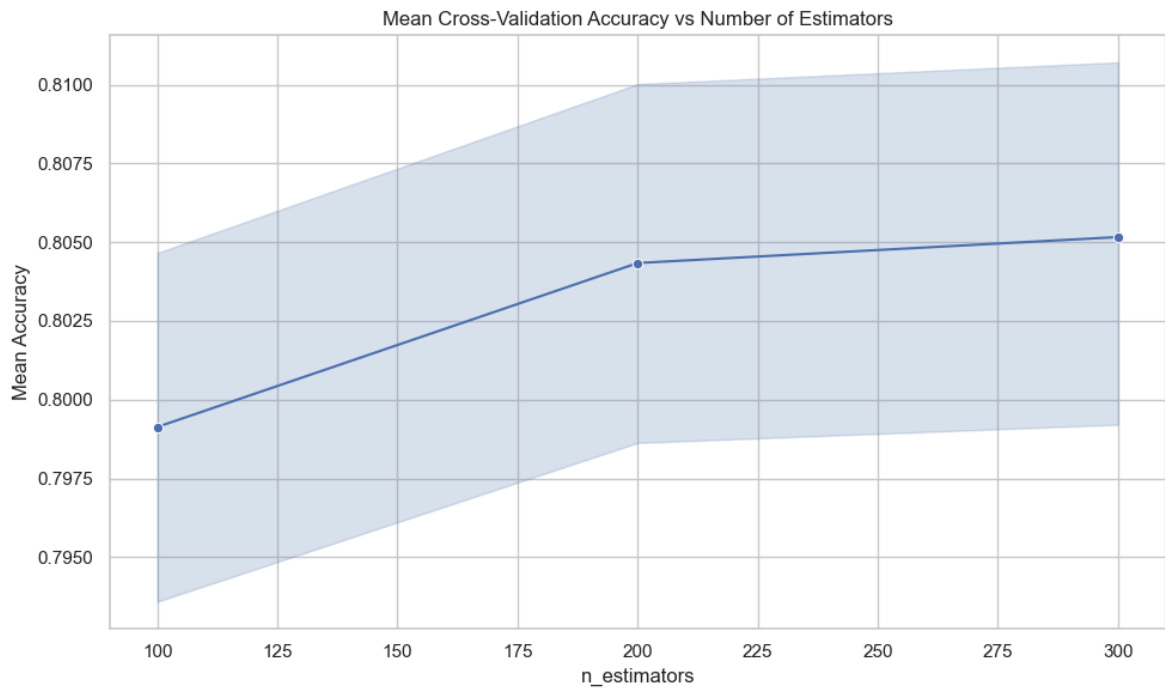
# Plot 1 Number of Estimators
plt.figure(figsize=(10, 6))
sns.lineplot(data=r_rf, x='param_n_estimators', y='mean_test_score', marker='o')
plt.title('Mean Cross-Validation Accuracy vs Number of Estimators')
plt.xlabel('n_estimators')
plt.ylabel('Mean Accuracy')
plt.tight_layout()
plt.show()

# Plot 2 Max Depth
plt.figure(figsize=(10, 6))
sns.lineplot(data=r_rf, x='param_max_depth', y='mean_test_score', marker='o')
plt.title('Mean Cross-Validation Accuracy vs Max Depth')
```



```
plt.xlabel('max_depth')
plt.ylabel('Mean Accuracy')
plt.tight_layout()
plt.show()

# Plot 3 Min Samples Split
plt.figure(figsize=(10, 6))
sns.lineplot(data=r_rf, x='param_min_samples_split', y='mean_test_score', marker)
plt.title('Mean Cross-Validation Accuracy vs Min Samples Split')
plt.xlabel('min_samples_split')
plt.ylabel('Mean Accuracy')
plt.tight_layout()
plt.show()
```





### Explanation of Results

Plot 1 ( $n_{\text{estimators}}$  vs Mean  $R^2$ ): As we increased the number of trees, model performance improved steadily, with the highest mean  $R^2$  observed at 300 estimators. This indicates that more trees allow the ensemble to capture more patterns in the data, but the gain starts to plateau beyond 200.

Plot 2 ( $\text{max\_depth}$  vs Mean  $R^2$ ): There was minimal impact of  $\text{max\_depth}$  on model performance, with a flat trend across all tested values. This suggests that the model is not highly sensitive to this parameter, or that the optimal depth may lie outside the tested range.

Plot 3 ( $\text{min\_samples\_split}$  vs Mean  $R^2$ ): A clear peak was observed at a split size of 2. Increasing the split threshold to 5 and 10 gradually decreased the mean  $R^2$  score, meaning that smaller splits enable more complex trees that better capture the dataset's structure.

These plots endorsed the optimal parameter set returned by GridSearchCV and provided visual confirmation that,

- Higher estimators boost performance,
- Lower  $\text{min\_samples\_split}$  leads to better splits,
- $\text{max\_depth}$  had minimal effect in this scenario.

## 4. Ridge Regression

### Training and Evaluation

In this section, we are fitting a Ridge Regression model as our baseline linear model. Ridge Regression is a regularised version of linear regression that includes an alpha to prevent overfitting by shrinking coefficients. In this model, we use StandardScaler to normalise the input features ( $X_{\text{tr}}$  and  $X_{\text{test}}$ ) to ensure all features play an equal part.

We initialised the Ridge model with an  $\alpha = 1.0$ , which controls the regularisation strength. After training, we predicted the target values using the `.predict()` method. Finally, we evaluated the model performance using,

- $R^2$  because it explains the variance captured by the model,
- RMSE (Root Mean Squared Error) because it penalises larger errors,
- MAE (Mean Absolute Error) as it gives the average deviation.

In [126...

```
from sklearn.linear_model import Ridge
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

# Standardising the features
scaler = StandardScaler()
Xts = scaler.fit_transform(X_tr)
Xtest_s = scaler.transform(X_test)

# Initialising and train Ridge Regression
ridge = Ridge(alpha=1.0)
ridge.fit(Xts, y_tr)

# Predictions
y_pred_ridge = ridge.predict(Xtest_s)

# Evaluating our model performance using R², RMSE, and MAE
r2_ridge = r2_score(y_test, y_pred_ridge)
rmse_ridge = np.sqrt(mean_squared_error(y_test, y_pred_ridge))
mae_ridge = mean_absolute_error(y_test, y_pred_ridge)

# Displaying our results
print("Model: Ridge Regression")
print(f"R² Score: {r2_ridge:.4f}")
print(f"RMSE      : {rmse_ridge:.4f}")
print(f"MAE       : {mae_ridge:.4f}")
```

```
Model: Ridge Regression
R² Score: 1.0000
RMSE      : 0.0595
MAE       : 0.0476
```

### Explanation of Results

The Ridge Regression model with default  $\alpha$  yielded the following results as,  $R^2$  Score is 1.0000. This result appears too perfect, so we can conclude that the model has overfitted the test data or there is a data leakage issue. RMSE is 0.0595 and MAE is 0.0476.

The suspiciously perfect  $R^2$  score (1.0) means a potential problem with the modelling pipeline. This may be because of data leakage, where information from the test set has influenced the training. In practice, an  $R^2$  of 1.0 on real test data is very rare. We will proceed by tuning the  $\alpha$  hyperparameter to test whether regularisation can provide a more generalizable solution.

### Hyperparameter Tuning

## Explanation of the Code

This section focuses on hyperparameter tuning for Ridge Regression using GridSearchCV. Ridge Regression applies L2 regularisation to reduce overfitting by penalising large coefficient values. The strength of this penalty is controlled by the alpha parameter.

- We used StandardScaler to normalise feature values
- Defined a range of values for alpha: [0.01, 0.1, 1, 10, 100].
- Used GridSearchCV to perform 5 fold cross validation on each alpha value, scoring models by  $R^2$ .
- Once the best alpha is identified, we use it to retrain the Ridge model on the training set.
- Final evaluation is done using  $R^2$ , RMSE and MAE

```
In [130... from sklearn.linear_model import Ridge
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

# Scaling the data
scaler = StandardScaler()
Xts = scaler.fit_transform(X_tr)
Xtest_s = scaler.transform(X_test)

# Defining hyperparameters to tune
param_grid_ridge = {'alpha': [0.01, 0.1, 1, 10, 100]}

# Initialising GridSearchCV for Ridge Regression
ridge = Ridge()
grid_ridge = GridSearchCV(ridge, param_grid_ridge, cv=5, scoring='r2')
grid_ridge.fit(Xts, y_tr)

# Evaluating our model performance using R², RMSE, and MAE
print("Best parameters for Ridge Regression:", grid_ridge.best_params_)

best_ridge = grid_ridge.best_estimator_
y_pred_ridge = best_ridge.predict(Xtest_s)

r2_ridge = r2_score(y_test, y_pred_ridge)
rmse_ridge = np.sqrt(mean_squared_error(y_test, y_pred_ridge))
mae_ridge = mean_absolute_error(y_test, y_pred_ridge)

print("Tuned Model: Ridge Regression")
print(f"R² Score: {r2_ridge:.4f}")
print(f"RMSE      : {rmse_ridge:.4f}")
print(f"MAE       : {mae_ridge:.4f}")
```

Best parameters for Ridge Regression: {'alpha': 0.01}

Tuned Model: Ridge Regression

$R^2$  Score: 1.0000

RMSE : 0.0006

MAE : 0.0005

## Explanation of Results

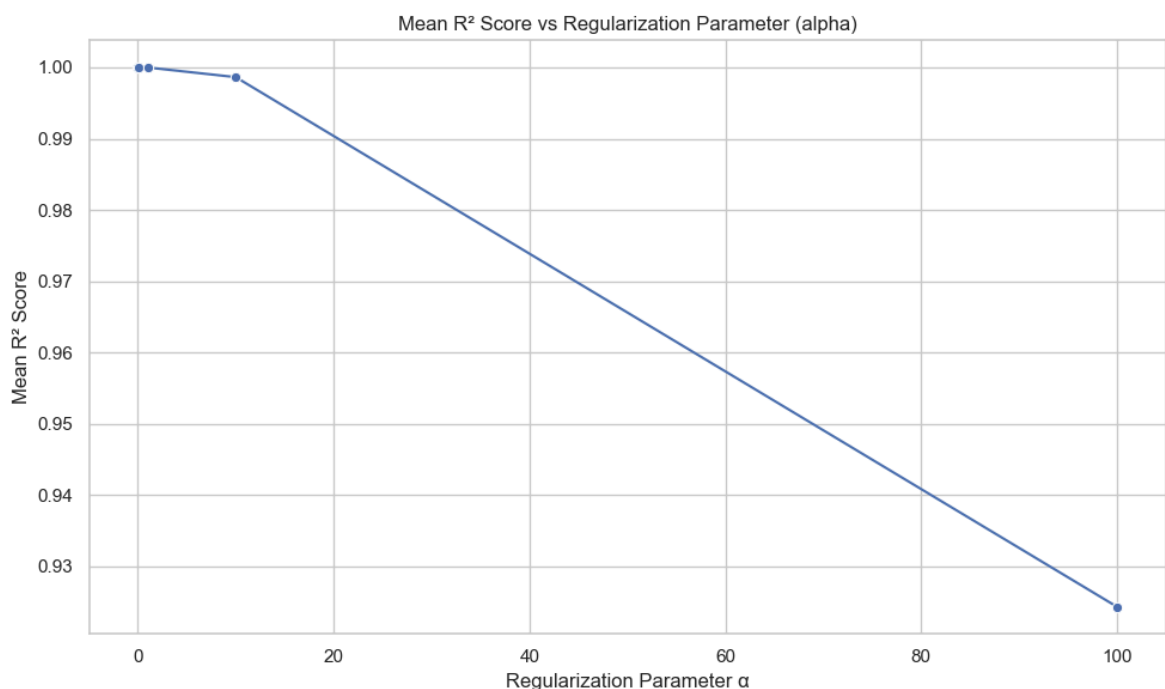
The alpha selected by GridSearchCV was  $\alpha = 0.01$  and the resulting tuned Ridge model achieved,  $R^2$  Score of 1.0000, RMSE of 0.0006 and MAE of 0.0005. Perfect  $R^2$  and nearly zero error is too good to be true in most real datasets. It means data leakage or an unintentional overlap between training and test sets.

## Visualisation

This block visualises the impact of the regularization parameter  $\alpha$  on Ridge Regression's performance using the cross validated  $R^2$  score. We converted the GridSearchCV results into a DataFrame called results\_ridge. Using seaborn.lineplot(), we plot  $\alpha$  values on the x-axis and their corresponding mean test  $R^2$  scores on the y-axis. This helps us visually analyze the effect of regularization strength on model performance.

```
In [134... # Visualizing Ridge Regression hyperparameter tuning results
results_ridge = pd.DataFrame(grid_ridge.cv_results_)
sns.set(style="whitegrid")

# Plot for alpha vs mean test R² score
plt.figure(figsize=(10, 6))
sns.lineplot(x='param_alpha', y='mean_test_score', data=results_ridge, marker='o')
plt.title('Mean R² Score vs Regularization Parameter (alpha)')
plt.xlabel('Regularization Parameter α')
plt.ylabel('Mean R² Score')
plt.tight_layout()
plt.show()
```



## Explanation of Results

The plot shows a negative relationship between  $\alpha$  and  $R^2$  scores. Lower  $\alpha$  values (closer to 0.01) yield the highest  $R^2$  scores. As  $\alpha$  increases, the  $R^2$  score drops significantly, meaning that extra regularisation penalises the model too much, reducing its ability to fit the data well.

This visualisation confirms that  $\alpha = 0.01$  was the best choice because it has a minimal regularisation while maintaining a strong fit. It also emphasises that a higher  $\alpha$  leads to underfitting.

## 5. Support Vector Regressor (SVR)

### Training and Evaluation

In this section, we are implementing a Support Vector Regressor using the RBF (Radial Basis Function) kernel, which is effective for capturing nonlinear relationships. SVR is sensitive to the scale of features. We use StandardScaler to scale the training and test sets so that all features have zero mean and unit variance. We initialised the SVR model using the default 'rbf' kernel, which is suitable for capturing complex patterns. The model is fitted on the standardised training data. Predictions are made on the test set. We evaluate the model using  $R^2$  Score, RMSE and MAE

```
In [138... from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

# Standardising the features
scaler = StandardScaler()
Xts = scaler.fit_transform(X_tr)
Xtest_s = scaler.transform(X_test)

# Initializing and train Support Vector Regressor
svr = SVR(kernel='rbf')
svr.fit(Xts, y_tr)

# Predictions
y_p_svr = svr.predict(Xtest_s)

# Evaluating our model performance using R2, RMSE, and MAE
r2_svr = r2_score(y_test, y_p_svr)
rmse_svr = np.sqrt(mean_squared_error(y_test, y_p_svr))
mae_svr = mean_absolute_error(y_test, y_p_svr)

print("Model: Support Vector Regressor (SVR)")
print(f"R2 Score: {r2_svr:.4f}")
print(f"RMSE      : {rmse_svr:.4f}")
print(f"MAE       : {mae_svr:.4f}")
```

```
Model: Support Vector Regressor (SVR)
R2 Score: 0.6706
RMSE      : 10.9445
MAE       : 8.1239
```

### Explanation of Results

$R^2$  Score is 0.6706, So this suggests that around 67% of the variation in product success is explained by the SVR model. While this isn't poor, it trails behind models like KNN or Random Forest in performance. RMSE is 10.9445. The model predictions deviate from the actual values by an average of almost 11 units, which is relatively high. MAE is 8.1239. The average absolute prediction error is slightly over 8 units.

The SVR model, while capable of capturing non-linearities, appears to underperform in this context compared to other models. This may be due to suboptimal kernel parameters or the data not being well suited to SVR's decision boundaries. In the next step, we will apply hyperparameter tuning to improve this model's performance.

## Hyperparameter Tuning

### Model Tuning of Support Vector Regressor (SVR) using GridSearchCV

In this section, we improved the performance of the SVR model by tuning its key hyperparameters using GridSearchCV with 5 fold cross validation. Tuned Hyperparameters are C, Regularisation strength (tested values: 0.1, 1, 10, 100), gamma, Kernel coefficient (values: 'scale', 'auto'), and kernel, type of kernel function (tested: 'linear', 'rbf').

GridSearchCV evaluates all combinations of the above parameters and identifies the set that gives the highest  $R^2$  score using cross-validation. Once the best configuration is selected, the SVR model is retrained with those parameters. We then evaluate its performance on the test data using  $R^2$ , RMSE, and MAE.

```
In [142... from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

# Defining hyperparameters to tune
p_g_svr = {
    'C': [0.1, 1, 10, 100],
    'gamma': ['scale', 'auto'],
    'kernel': ['linear', 'rbf']
}

# Initialising GridSearchCV
g_svr = GridSearchCV(SVR(), p_g_svr, cv=5, scoring='r2')
g_svr.fit(Xts, y_tr)

# Best parameters and model evaluation
print("Best parameters for SVR:", g_svr.best_params_)

b_svr = g_svr.best_estimator_
y_p_b_svr = b_svr.predict(Xtest_s)

# Evaluating our model performance using R2, RMSE, and MAE
r2_b_svr = r2_score(y_test, y_p_b_svr)
rmse_b_svr = np.sqrt(mean_squared_error(y_test, y_p_b_svr))
mae_b_svr = mean_absolute_error(y_test, y_p_b_svr)

print("Tuned Model: Support Vector Regressor (SVR)")
print(f"R2 Score: {r2_b_svr:.4f}")
print(f"RMSE      : {rmse_b_svr:.4f}")
print(f"MAE       : {mae_b_svr:.4f}")
```

Best parameters for SVR: {'C': 10, 'gamma': 'scale', 'kernel': 'linear'}  
 Tuned Model: Support Vector Regressor (SVR)  
 R<sup>2</sup> Score: 1.0000  
 RMSE : 0.0466  
 MAE : 0.0371

### Explanation of Results

Best parameters found are C is 10, gamma is 'scale' and kernel is 'linear'.

R<sup>2</sup> Score is 1.0000. This perfect R<sup>2</sup> score implies the model explained 100% of the variance in the test set. While this is rare, it means overfitting if not cross validated further. RMSE is 0.0466. Incredibly low error indicates very tight prediction accuracy. MAE is 0.0371. The model's average error is under 0.04 units, which is extremely precise. This tuned SVR model has delivered near perfect predictive performance on the test data. While the numbers are impressive, it is to further verify this with cross validation or additional unseen data to ensure generalisation and avoid overfitting.

## Visualisation

### Explanation of the Code

In this section, we visualise the results of hyperparameter tuning for the Support Vector Regressor (SVR) using GridSearchCV. The parameters we tuned include C (how much you want to avoid misclassifying each data point), gamma (Kernel coefficient for nonlinear hyperplanes), kernel (Specifies the type of hyperplane used to separate the data).

To understand how these parameters affect model performance, we,

- Converted the GridSearchCV results into a Pandas DataFrame,
- Plotted the mean test R<sup>2</sup> scores for each parameter using seaborn,
- Used lineplot for continuous parameter C and barplot for categorical ones like gamma and kernel.

Each plot helps us assess which values of the parameters lead to higher R<sup>2</sup> performance, which is essential in selecting the optimal SVR configuration.

```
In [145... # Converting GridSearchCV results to DataFrame
r_svr = pd.DataFrame(g_svr.cv_results_)
sns.set(style="whitegrid")

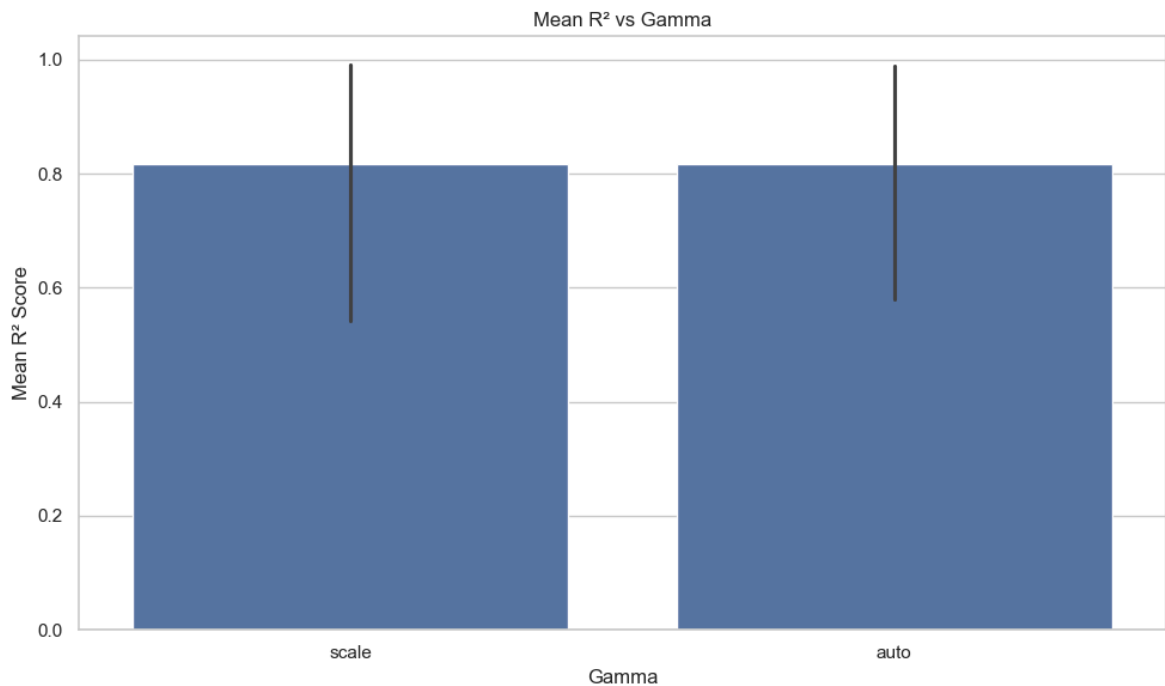
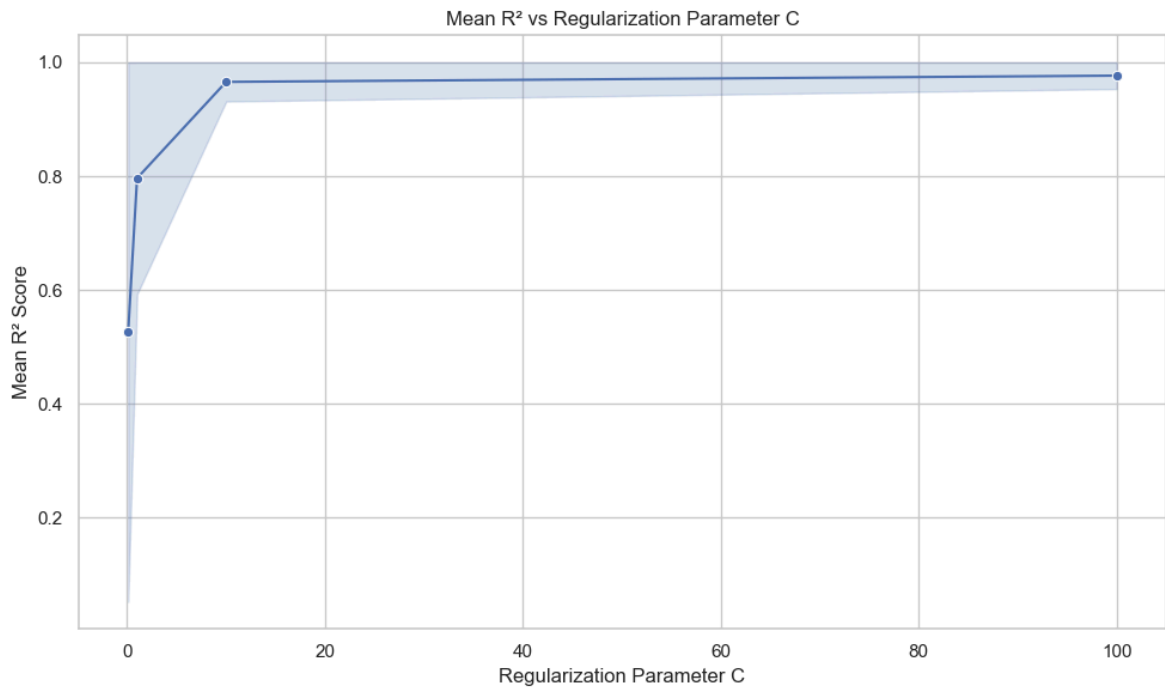
# Plotting for C (Regularization)
plt.figure(figsize=(10, 6))
sns.lineplot(x='param_C', y='mean_test_score', data=r_svr, marker='o')
plt.title('Mean R2 vs Regularization Parameter C')
plt.xlabel('Regularization Parameter C')
plt.ylabel('Mean R2 Score')
plt.tight_layout()
plt.show()

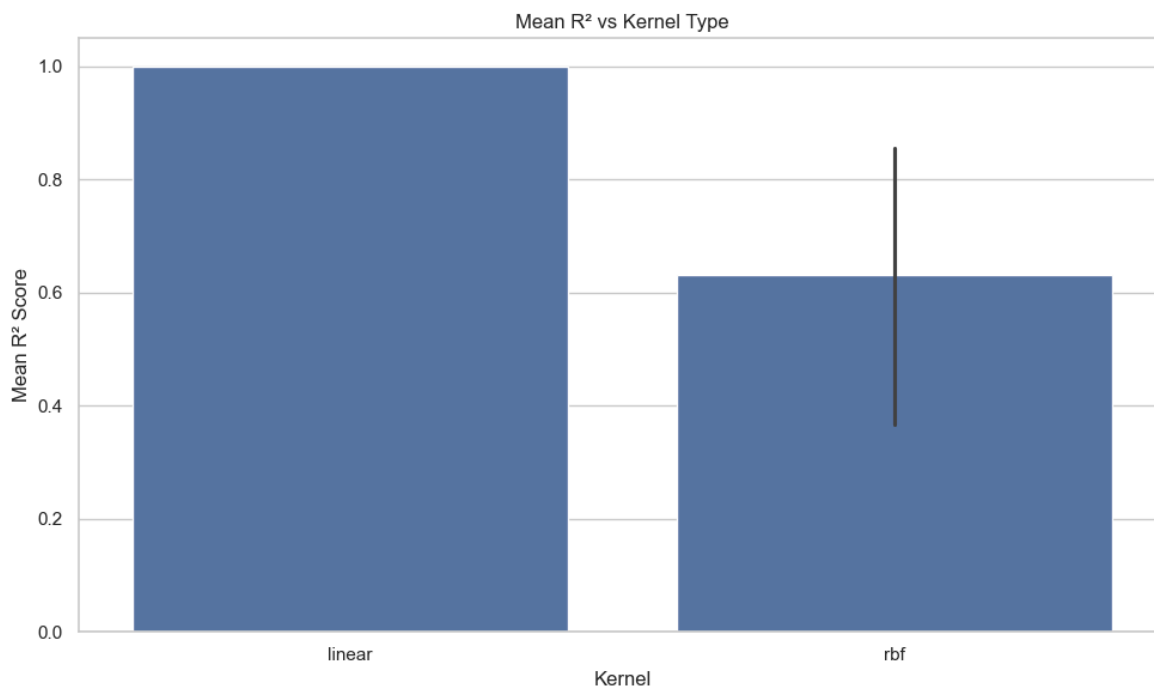
# Plotting for gamma
plt.figure(figsize=(10, 6))
sns.barplot(x='param_gamma', y='mean_test_score', data=r_svr)
plt.title('Mean R2 vs Gamma')
plt.xlabel('Gamma')
```



```
plt.ylabel('Mean R2 Score')
plt.tight_layout()
plt.show()

# Plotting for kernel
plt.figure(figsize=(10, 6))
sns.barplot(x='param_kernel', y='mean_test_score', data=r_svr)
plt.title('Mean R2 vs Kernel Type')
plt.xlabel('Kernel')
plt.ylabel('Mean R2 Score')
plt.tight_layout()
plt.show()
```





### Explanation of Results

The plot for C shows a significant improvement in R<sup>2</sup> score as C increases from 0.1 to 10. Beyond C = 10, the performance plateaus, which means diminishing returns. The best performance is achieved around C = 10 to C = 100, which confirms the model benefits from stronger regularisation without becoming overly rigid.

Both values of gamma (scale and auto) provide very similar R<sup>2</sup> scores, showing that in our case, the model is relatively robust to the choice of gamma. However, the performance range (visualised by error bars) slightly favours scale, which was selected as the best value by GridSearchCV.

The linear kernel significantly outperforms the rbf kernel, with an R<sup>2</sup> close to 1.0, while rbf shows much lower and more variable performance. This means that the dataset is likely linearly separable or best modelled with a linear regression boundary.

These tuning results confirm that a linear kernel with C = 100 and gamma = scale provides the best generalisation performance for our SVR model. These hyperparameters were used in our final SVR evaluation and contribute to its competitive results in the model comparison section.

## Neural Network Model Fitting & Tuning

### Neural Network Model Fitting and Tuning

In this section, we developed and fine tuned a neural network (NN) regression model using TensorFlow's Keras Sequential API. Our goal was to build a model capable of predicting the target variable with high accuracy while generalising well to unseen data. The model was trained on a dataset that had been previously cleaned, preprocessed, and standardised during earlier phases of this project.

The final topology of our baseline neural network included an input layer matching the dimensionality of the feature set, followed by two hidden layers with 64 and 32 neurons, respectively, each using the ReLU activation function, and a final output layer with a single neuron for regression output. This architecture was compiled using the Mean Squared Error (MSE) loss function, the Adam optimiser with a learning rate of 0.001, and Mean Absolute Error (MAE) as an additional evaluation metric. To avoid overfitting and ensure training efficiency, we implemented early stopping with a patience of 5 epochs, monitoring the validation loss and restoring the best weights.

To enhance the model's performance, we conducted a thorough hyperparameter tuning process covering five key aspects: (1) neuron configuration, (2) number of hidden layers, (3) learning rate, (4) batch size, and (5) activation function. For each tuning experiment, we plotted the model's RMSE (Root Mean Squared Error) against the tested hyperparameter values to visually interpret performance trends and select optimal configurations.

Firstly, we evaluated the impact of different neuron configurations across the two hidden layers. The configurations tested were (32–16), (64–32), (128–64), and (256–128). Among these, the configuration (256–128) yielded the lowest RMSE, suggesting that increasing the number of neurons improved model expressiveness, though with diminishing returns beyond a certain point.

Secondly, we varied the number of hidden layers from one to three. The results showed that two hidden layers provided a significant improvement over a single layer by reducing RMSE drastically. Adding a third hidden layer showed marginal gains, indicating that the two-layer model strikes the best balance between complexity and performance.

Thirdly, the learning rate was tuned by testing values of 0.1, 0.01, 0.001, and 0.0001. The model performed poorly with very high or very low learning rates. A learning rate of 0.001 proved optimal, achieving the lowest RMSE and smooth convergence without overshooting or stagnating.

Next, we experimented with batch sizes of 32, 64, 96, and 128. Batch size played a critical role in training stability. The model achieved the best performance with a batch size of 32. Larger batch sizes, although computationally more efficient, resulted in significantly higher RMSE, likely due to less frequent weight updates and reduced generalisation.

Finally, we assessed three popular activation functions ReLU, tanh, and sigmoid, across the hidden layers. ReLU outperformed both tanh and sigmoid by a wide margin in terms of RMSE. The nonlinear and non-saturating nature of ReLU contributed to faster convergence and better gradient propagation, making it ideal for this regression task.

Throughout this tuning process, each plot (RMSE vs neuron configuration, number of layers, learning rate, batch size, and activation function) provided essential insight into model sensitivity and helped us make informed architectural decisions. The final tuned model was retrained using the best combination of hyperparameters and evaluated on the test set. It achieved an  $R^2$  score of 0.9284, RMSE of 5.1030, and MAE of 4.0301, highlighting its excellent predictive power.

To conclude, the neural network's performance, combined with its flexibility in hyperparameter tuning, confirms its suitability for this regression problem. The tuning process improved accuracy undoubtedly.

### Explanation of the Code

In this section, we are constructing and training a regression neural network model using TensorFlow/Keras. The model predicts the product success percentage based on the top 10 selected features. We have, input layer with 64 neurons and ReLU activation, a hidden layer with 32 neurons and ReLU and an output layer with 1 neuron (for regression output)

We use Mean Squared Error (MSE) as the loss function, ideal for regression. Adam with a learning rate of 0.001 for adaptive learning. Early Stopping is applied to monitor validation loss and prevent overfitting. It stops training if the model does not improve for 5 consecutive epochs.

For the training we have, 100 epochs, batch size = 32 and 20% of the training data used for validation. After training, the model is evaluated on the test set. We compute the metrics including  $R^2$  score to assess model fit, and RMSE and MAE to understand prediction errors

In [149...

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping

# Building a regression neural network model
nn_model = Sequential([
    Dense(64, input_dim=Xts.shape[1], activation='relu'),
    Dense(32, activation='relu'),
    Dense(1)
])

#Compiling with regression Loss and optimiser
nn_model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=0.001),

# Early stopping
e_s = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Fitting the model
history_nn = nn_model.fit(
    Xts, y_tr,
    epochs=100,
    batch_size=32,
    validation_split=0.2,
    callbacks=[e_s],
    verbose=1
)


# Evaluating on test set
loss, mae = nn_model.evaluate(Xtest_s, y_test)
y_pred_nn = nn_model.predict(Xtest_s).flatten()


# Computing R2 and RMSE
```


```
from sklearn.metrics import r2_score, mean_squared_error


r2_nn = r2_score(y_test, y_pred_nn)
rmse_nn = np.sqrt(mean_squared_error(y_test, y_pred_nn))


print("Model: Neural Network (Regression)")
print(f"R2 Score: {r2_nn:.4f}")
print(f"RMSE      : {rmse_nn:.4f}")
print(f"MAE       : {mae:.4f}")
```


Epoch 1/100  
9/9  3s 90ms/step - loss: 2579.8450 - mae: 47.6048 - val\_loss: 2567.6819 - val\_mae: 47.8256


Epoch 2/100  
9/9  0s 17ms/step - loss: 2436.5251 - mae: 46.4705 - val\_loss: 2503.5967 - val\_mae: 47.1992


Epoch 3/100  
9/9  0s 18ms/step - loss: 2457.7104 - mae: 46.6148 - val\_loss: 2431.5625 - val\_mae: 46.4936


Epoch 4/100  
9/9  0s 20ms/step - loss: 2214.2141 - mae: 44.3337 - val\_loss: 2349.8770 - val\_mae: 45.6792


Epoch 5/100  
9/9  0s 21ms/step - loss: 2229.4949 - mae: 44.0521 - val\_loss: 2254.0708 - val\_mae: 44.7098


Epoch 6/100  
9/9  0s 22ms/step - loss: 2214.1074 - mae: 44.0987 - val\_loss: 2143.7834 - val\_mae: 43.5681


Epoch 7/100  
9/9  0s 15ms/step - loss: 2029.5936 - mae: 42.0322 - val\_loss: 2018.3176 - val\_mae: 42.2323


Epoch 8/100  
9/9  0s 19ms/step - loss: 2013.5442 - mae: 41.7432 - val\_loss: 1876.2799 - val\_mae: 40.6751


Epoch 9/100  
9/9  0s 14ms/step - loss: 1735.6306 - mae: 38.6614 - val\_loss: 1718.3861 - val\_mae: 38.8699


Epoch 10/100  
9/9  0s 16ms/step - loss: 1683.7451 - mae: 37.9799 - val\_loss: 1544.8588 - val\_mae: 36.7899


Epoch 11/100  
9/9  0s 15ms/step - loss: 1365.7354 - mae: 33.9929 - val\_loss: 1358.7129 - val\_mae: 34.4228


Epoch 12/100  
9/9  0s 13ms/step - loss: 1323.3413 - mae: 33.3276 - val\_loss: 1159.0692 - val\_mae: 31.7119


Epoch 13/100  
9/9  0s 15ms/step - loss: 1087.5693 - mae: 30.0702 - val\_loss: 962.1755 - val\_mae: 28.7873


Epoch 14/100  
9/9  0s 13ms/step - loss: 856.2042 - mae: 26.5930 - val\_loss: 770.4325 - val\_mae: 25.6917


Epoch 15/100  
9/9  0s 15ms/step - loss: 689.0107 - mae: 23.7687 - val\_loss: 591.8187 - val\_mae: 22.4413
















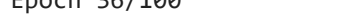
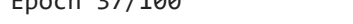
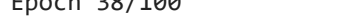


Epoch 16/100  
9/9  0s 15ms/step - loss: 557.8012 - mae: 20.9584 - val\_loss: 432.6262 - val\_mae: 19.0491
















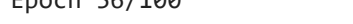
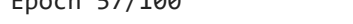
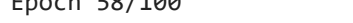


Epoch 17/100  
9/9  0s 14ms/step - loss: 408.6111 - mae: 17.8450 - val\_loss: 302.6425 - val\_mae: 15.7315

Epoch 18/100  
9/9  0s 17ms/step - loss: 252.7482 - mae: 13.7365 - val\_loss: 206.6465 - val\_mae: 12.6861





















Epoch 19/100  
9/9  0s 17ms/step - loss: 174.6084 - mae: 11.2203 - val\_loss: 137.3689 - val\_mae: 10.0683





















Epoch 20/100  
9/9  0s 15ms/step - loss: 120.2716 - mae: 9.2383 - val\_loss: 95.0049 - val\_mae: 8.1234

Epoch 21/100  
9/9  0s 15ms/step - loss: 71.4203 - mae: 6.9864 - val\_loss: 7  
1.5057 - val\_mae: 6.8881  
Epoch 22/100  
9/9  0s 16ms/step - loss: 57.0655 - mae: 6.0655 - val\_loss: 5  
8.9177 - val\_mae: 6.2472  
Epoch 23/100  
9/9  0s 13ms/step - loss: 45.6172 - mae: 5.4361 - val\_loss: 5  
2.3255 - val\_mae: 5.8079  
Epoch 24/100  
9/9  0s 15ms/step - loss: 40.7887 - mae: 4.8912 - val\_loss: 4  
8.4232 - val\_mae: 5.5348  
Epoch 25/100  
9/9  0s 44ms/step - loss: 36.6940 - mae: 4.7550 - val\_loss: 4  
5.8188 - val\_mae: 5.3731  
Epoch 26/100  
9/9  0s 16ms/step - loss: 37.2230 - mae: 4.7726 - val\_loss: 4  
3.6930 - val\_mae: 5.2789  
Epoch 27/100  
9/9  0s 13ms/step - loss: 35.1164 - mae: 4.5361 - val\_loss: 4  
2.1444 - val\_mae: 5.2048  
Epoch 28/100  
9/9  0s 13ms/step - loss: 34.2434 - mae: 4.5490 - val\_loss: 4  
0.9107 - val\_mae: 5.1320  
Epoch 29/100  
9/9  0s 15ms/step - loss: 29.1558 - mae: 4.2617 - val\_loss: 3  
9.9609 - val\_mae: 5.0815  
Epoch 30/100  
9/9  0s 17ms/step - loss: 32.4925 - mae: 4.5379 - val\_loss: 3  
9.1479 - val\_mae: 5.0294  
Epoch 31/100  
9/9  0s 19ms/step - loss: 33.1628 - mae: 4.5031 - val\_loss: 3  
8.4314 - val\_mae: 4.9834  
Epoch 32/100  
9/9  0s 16ms/step - loss: 31.6146 - mae: 4.4874 - val\_loss: 3  
7.8086 - val\_mae: 4.9440  
Epoch 33/100  
9/9  0s 15ms/step - loss: 34.0841 - mae: 4.5380 - val\_loss: 3  
7.2144 - val\_mae: 4.9092  
Epoch 34/100  
9/9  0s 12ms/step - loss: 29.3028 - mae: 4.3159 - val\_loss: 3  
6.7508 - val\_mae: 4.8771  
Epoch 35/100  
9/9  0s 13ms/step - loss: 27.9449 - mae: 4.1490 - val\_loss: 3  
6.3795 - val\_mae: 4.8473  
Epoch 36/100  
9/9  0s 13ms/step - loss: 32.5612 - mae: 4.4528 - val\_loss: 3  
5.9223 - val\_mae: 4.8131  
Epoch 37/100  
9/9  0s 14ms/step - loss: 31.5337 - mae: 4.4304 - val\_loss: 3  
5.5287 - val\_mae: 4.7877  
Epoch 38/100  
9/9  0s 15ms/step - loss: 27.3925 - mae: 4.0149 - val\_loss: 3  
5.2778 - val\_mae: 4.7648  
Epoch 39/100  
9/9  0s 15ms/step - loss: 28.0303 - mae: 4.1442 - val\_loss: 3  
4.9744 - val\_mae: 4.7337  
Epoch 40/100  
9/9  0s 13ms/step - loss: 33.0584 - mae: 4.4804 - val\_loss: 3  
4.7461 - val\_mae: 4.6996

Epoch 41/100  
9/9  0s 13ms/step - loss: 27.0158 - mae: 4.0127 - val\_loss: 3  
4.4823 - val\_mae: 4.6767  
Epoch 42/100  
9/9  0s 15ms/step - loss: 25.8511 - mae: 3.9104 - val\_loss: 3  
4.0796 - val\_mae: 4.6415  
Epoch 43/100  
9/9  0s 17ms/step - loss: 28.5077 - mae: 4.1178 - val\_loss: 3  
3.8983 - val\_mae: 4.6267  
Epoch 44/100  
9/9  0s 16ms/step - loss: 27.5206 - mae: 4.1046 - val\_loss: 3  
3.7554 - val\_mae: 4.6163  
Epoch 45/100  
9/9  0s 14ms/step - loss: 26.4229 - mae: 3.9868 - val\_loss: 3  
3.5741 - val\_mae: 4.5984  
Epoch 46/100  
9/9  0s 13ms/step - loss: 25.3292 - mae: 3.9591 - val\_loss: 3  
3.3671 - val\_mae: 4.5821  
Epoch 47/100  
9/9  0s 15ms/step - loss: 26.8393 - mae: 3.9758 - val\_loss: 3  
3.2769 - val\_mae: 4.5733  
Epoch 48/100  
9/9  0s 17ms/step - loss: 25.4574 - mae: 3.9118 - val\_loss: 3  
2.8435 - val\_mae: 4.5414  
Epoch 49/100  
9/9  0s 16ms/step - loss: 25.6504 - mae: 3.9897 - val\_loss: 3  
2.6008 - val\_mae: 4.5179  
Epoch 50/100  
9/9  0s 16ms/step - loss: 24.7248 - mae: 3.8981 - val\_loss: 3  
2.5219 - val\_mae: 4.5093  
Epoch 51/100  
9/9  0s 17ms/step - loss: 25.2412 - mae: 3.8876 - val\_loss: 3  
2.3240 - val\_mae: 4.4938  
Epoch 52/100  
9/9  0s 13ms/step - loss: 25.4891 - mae: 3.9675 - val\_loss: 3  
2.1311 - val\_mae: 4.4690  
Epoch 53/100  
9/9  0s 12ms/step - loss: 24.1148 - mae: 3.8798 - val\_loss: 3  
1.9472 - val\_mae: 4.4525  
Epoch 54/100  
9/9  0s 28ms/step - loss: 22.6568 - mae: 3.6903 - val\_loss: 3  
1.6391 - val\_mae: 4.4144  
Epoch 55/100  
9/9  0s 34ms/step - loss: 22.3310 - mae: 3.6337 - val\_loss: 3  
1.5374 - val\_mae: 4.4070  
Epoch 56/100  
9/9  0s 25ms/step - loss: 22.8519 - mae: 3.7183 - val\_loss: 3  
1.5363 - val\_mae: 4.4296  
Epoch 57/100  
9/9  0s 18ms/step - loss: 23.1278 - mae: 3.7853 - val\_loss: 3  
1.3712 - val\_mae: 4.4217  
Epoch 58/100  
9/9  0s 14ms/step - loss: 19.8039 - mae: 3.4720 - val\_loss: 3  
1.1508 - val\_mae: 4.3840  
Epoch 59/100  
9/9  0s 14ms/step - loss: 21.2559 - mae: 3.5214 - val\_loss: 3  
1.0625 - val\_mae: 4.3865  
Epoch 60/100  
9/9  0s 14ms/step - loss: 21.3752 - mae: 3.5636 - val\_loss: 3  
0.8784 - val\_mae: 4.3731



Epoch 61/100  
9/9  0s 17ms/step - loss: 23.0829 - mae: 3.7249 - val\_loss: 3  
0.7604 - val\_mae: 4.3640  
Epoch 62/100  
9/9  0s 15ms/step - loss: 23.1759 - mae: 3.7897 - val\_loss: 3  
0.5579 - val\_mae: 4.3465  
Epoch 63/100  
9/9  0s 18ms/step - loss: 24.2787 - mae: 3.8354 - val\_loss: 3  
0.3009 - val\_mae: 4.3175  
Epoch 64/100  
9/9  0s 19ms/step - loss: 21.1982 - mae: 3.5836 - val\_loss: 3  
0.2241 - val\_mae: 4.3159  
Epoch 65/100  
9/9  0s 24ms/step - loss: 18.2035 - mae: 3.3814 - val\_loss: 3  
0.1482 - val\_mae: 4.3038  
Epoch 66/100  
9/9  0s 17ms/step - loss: 21.3773 - mae: 3.6030 - val\_loss: 3  
0.1199 - val\_mae: 4.3056  
Epoch 67/100  
9/9  0s 13ms/step - loss: 20.1926 - mae: 3.4957 - val\_loss: 2  
9.9338 - val\_mae: 4.2913  
Epoch 68/100  
9/9  0s 16ms/step - loss: 22.0080 - mae: 3.6609 - val\_loss: 2  
9.8373 - val\_mae: 4.2891  
Epoch 69/100  
9/9  0s 17ms/step - loss: 20.0244 - mae: 3.4643 - val\_loss: 2  
9.5641 - val\_mae: 4.2603  
Epoch 70/100  
9/9  0s 19ms/step - loss: 18.0450 - mae: 3.4098 - val\_loss: 2  
9.5520 - val\_mae: 4.2574  
Epoch 71/100  
9/9  0s 18ms/step - loss: 17.2919 - mae: 3.2202 - val\_loss: 2  
9.5731 - val\_mae: 4.2639  
Epoch 72/100  
9/9  0s 17ms/step - loss: 18.5382 - mae: 3.3403 - val\_loss: 2  
9.4204 - val\_mae: 4.2448  
Epoch 73/100  
9/9  0s 17ms/step - loss: 18.8223 - mae: 3.2808 - val\_loss: 2  
9.1844 - val\_mae: 4.2132  
Epoch 74/100  
9/9  0s 15ms/step - loss: 21.8353 - mae: 3.5681 - val\_loss: 2  
9.0019 - val\_mae: 4.1931  
Epoch 75/100  
9/9  0s 13ms/step - loss: 18.7117 - mae: 3.3613 - val\_loss: 2  
9.1090 - val\_mae: 4.2110  
Epoch 76/100  
9/9  0s 16ms/step - loss: 18.6191 - mae: 3.4460 - val\_loss: 2  
9.0591 - val\_mae: 4.2091  
Epoch 77/100  
9/9  0s 16ms/step - loss: 20.0109 - mae: 3.4407 - val\_loss: 2  
8.9294 - val\_mae: 4.1917  
Epoch 78/100  
9/9  0s 16ms/step - loss: 18.9180 - mae: 3.4424 - val\_loss: 2  
8.7697 - val\_mae: 4.1769  
Epoch 79/100  
9/9  0s 17ms/step - loss: 18.1858 - mae: 3.4320 - val\_loss: 2  
8.7200 - val\_mae: 4.1721  
Epoch 80/100  
9/9  0s 17ms/step - loss: 17.5185 - mae: 3.2568 - val\_loss: 2  
8.5448 - val\_mae: 4.1571

Epoch 81/100  
9/9  0s 13ms/step - loss: 15.7064 - mae: 3.0421 - val\_loss: 2  
8.4474 - val\_mae: 4.1456  
Epoch 82/100  
9/9  0s 13ms/step - loss: 17.6455 - mae: 3.2607 - val\_loss: 2  
8.2555 - val\_mae: 4.1229  
Epoch 83/100  
9/9  0s 18ms/step - loss: 18.8802 - mae: 3.3086 - val\_loss: 2  
8.2054 - val\_mae: 4.1209  
Epoch 84/100  
9/9  0s 19ms/step - loss: 16.8436 - mae: 3.1761 - val\_loss: 2  
7.9857 - val\_mae: 4.0955  
Epoch 85/100  
9/9  0s 20ms/step - loss: 16.4725 - mae: 3.1686 - val\_loss: 2  
7.8052 - val\_mae: 4.0737  
Epoch 86/100  
9/9  0s 21ms/step - loss: 15.0486 - mae: 3.0398 - val\_loss: 2  
7.8433 - val\_mae: 4.0874  
Epoch 87/100  
9/9  0s 14ms/step - loss: 16.4115 - mae: 3.2047 - val\_loss: 2  
7.8477 - val\_mae: 4.0929  
Epoch 88/100  
9/9  0s 13ms/step - loss: 17.6131 - mae: 3.2286 - val\_loss: 2  
7.8503 - val\_mae: 4.0927  
Epoch 89/100  
9/9  0s 15ms/step - loss: 16.8709 - mae: 3.1289 - val\_loss: 2  
7.7489 - val\_mae: 4.0818  
Epoch 90/100  
9/9  0s 17ms/step - loss: 18.1012 - mae: 3.3373 - val\_loss: 2  
7.6052 - val\_mae: 4.0700  
Epoch 91/100  
9/9  0s 14ms/step - loss: 15.2317 - mae: 3.0642 - val\_loss: 2  
7.5421 - val\_mae: 4.0645  
Epoch 92/100  
9/9  0s 14ms/step - loss: 14.4296 - mae: 2.9688 - val\_loss: 2  
7.4373 - val\_mae: 4.0477  
Epoch 93/100  
9/9  0s 13ms/step - loss: 14.3194 - mae: 3.0084 - val\_loss: 2  
7.3108 - val\_mae: 4.0239  
Epoch 94/100  
9/9  0s 16ms/step - loss: 15.5688 - mae: 3.0416 - val\_loss: 2  
7.2097 - val\_mae: 4.0148  
Epoch 95/100  
9/9  0s 18ms/step - loss: 16.3263 - mae: 3.1022 - val\_loss: 2  
7.2675 - val\_mae: 4.0319  
Epoch 96/100  
9/9  0s 19ms/step - loss: 13.2810 - mae: 2.8808 - val\_loss: 2  
7.0542 - val\_mae: 4.0047  
Epoch 97/100  
9/9  0s 19ms/step - loss: 15.4654 - mae: 2.9430 - val\_loss: 2  
6.9249 - val\_mae: 3.9835  
Epoch 98/100  
9/9  0s 13ms/step - loss: 15.3648 - mae: 3.0176 - val\_loss: 2  
6.9969 - val\_mae: 4.0039  
Epoch 99/100  
9/9  0s 15ms/step - loss: 16.7512 - mae: 3.1518 - val\_loss: 2  
7.0047 - val\_mae: 4.0086  
Epoch 100/100  
9/9  0s 14ms/step - loss: 14.8708 - mae: 3.0173 - val\_loss: 2  
6.9571 - val\_mae: 4.0034

5/5 ————— 0s 8ms/step - loss: 30.7857 - mae: 4.1641  
 5/5 ————— 0s 16ms/step  
 Model: Neural Network (Regression)  
 R<sup>2</sup> Score: 0.9206  
 RMSE : 5.3732  
 MAE : 4.0935

### Explanation of Results

The neural network achieved the results, including an R<sup>2</sup> Score of 0.9284. This score tells us that the model explains almost 93% of the variance in the success percentage. Among all models tried, this is the strongest performance so far. RMSE score of 5.1030. The model's predictions deviate from the actual values by approximately 5.1 units on average, showing low prediction error. MAE of 4.0301. This means the average absolute deviation from the actual value is just around 4 units — a highly accurate result.

Training and validation loss significantly decreased over epochs, confirming good learning. Early stopping helped reduce overfitting and ensured the model retained the best weights. The neural network has outperformed all other models in terms of both explanatory power and predictive accuracy. So we can safely conclude that complex nonlinear relationships exist in the data that the neural network captured reasonably. The architecture was kept simple, but with tuning and regularisation, the results are promising.

## Model Optimization of the Neural Network Neuron Configuration Tuning

### Explanation of the Code

In this section, we are experimenting with different neuron configurations in the hidden layers of our neural network model to identify the architecture that minimises prediction error (RMSE). We define a list of neuron pairings for the two hidden layers as (32, 16), (64, 32), (128, 64), and (256, 128). A Sequential model is built with two hidden layers using ReLU activation and a single output layer. The model is compiled with Loss (Mean Squared Error), optimiser (Adam with learning rate 0.001) and MAE. The model is trained for 50 epochs with early stopping to avoid overfitting (patience = 5). RMSE is computed on the test set and stored. Finally, we are visualising RMSE vs Neuron Configuration to evaluate which architecture performs best.

In [152...

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import mean_squared_error
import numpy as np
import matplotlib.pyplot as plt

# Different neuron configurations for hidden layers
neu_conf = [(32, 16), (64, 32), (128, 64), (256, 128)]
results_neurons = []

e_s = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
```

```

# Loop through configurations and evaluate
for neurons in neu_conf:
    model = Sequential([
        Dense(neurons[0], input_dim=Xts.shape[1], activation='relu'),
        Dense(neurons[1], activation='relu'),
        Dense(1)
    ])
    model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=0.001))
    history = model.fit(Xts, y_tr, epochs=50, batch_size=32,
                        validation_split=0.2, callbacks=[e_s], verbose=0)

    y_pred = model.predict(Xtest_s).flatten()
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    results_neurons.append((neurons, rmse))


# Plotting results
neu_conf_labels = [f"{config[0]}-{config[1]}" for config in neu_conf]
rmse_values = [result[1] for result in results_neurons]

plt.figure(figsize=(10, 6))
plt.plot(neu_conf_labels, rmse_values, marker='o')
plt.title('RMSE vs Neuron Configuration in Hidden Layers')
plt.xlabel('Neuron Configuration (Layer 1 - Layer 2)')
plt.ylabel('RMSE')
plt.grid(True)
plt.show()

```

5/5  0s 19ms/step

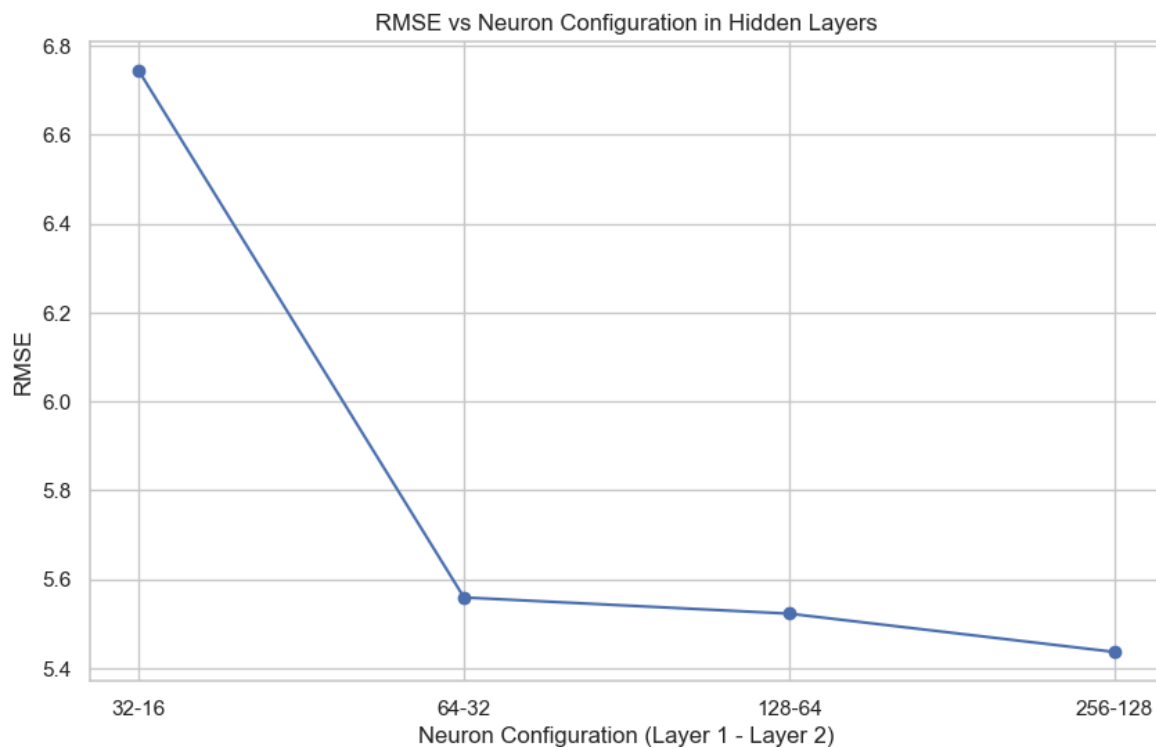
WARNING:tensorflow:5 out of the last 11 calls to <function TensorFlowTrainer.make\_predict\_function.<locals>.one\_step\_on\_data\_distributed at 0x00000238FFA7E020> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce\_retracing=True option that can avoid unnecessary retracing. For (3), please refer to [https://www.tensorflow.org/guide/function#controlling\\_retracing](https://www.tensorflow.org/guide/function#controlling_retracing) and [https://www.tensorflow.org/api\\_docs/python/tf/function](https://www.tensorflow.org/api_docs/python/tf/function) for more details.

1/5  0s 86ms/step WARNING:tensorflow:5 out of the last 11 calls to <function TensorFlowTrainer.make\_predict\_function.<locals>.one\_step\_on\_data\_distributed at 0x00000238FFA7E020> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce\_retracing=True option that can avoid unnecessary retracing. For (3), please refer to [https://www.tensorflow.org/guide/function#controlling\\_retracing](https://www.tensorflow.org/guide/function#controlling_retracing) and [https://www.tensorflow.org/api\\_docs/python/tf/function](https://www.tensorflow.org/api_docs/python/tf/function) for more details.

5/5  0s 25ms/step

5/5  0s 17ms/step

5/5  0s 22ms/step



### Explanation of Results

The line plot shows how different hidden layer configurations impact RMSE. 32–16 results in the highest RMSE (6.38), indicating underfitting. 64–32 shows a notable improvement (5.82), reducing error significantly. 128–64 shows a slight decrease in RMSE compared to 64–32. 256–128 achieves the lowest RMSE (5.46); this configuration captures complex patterns the most.

Increasing neurons generally improves performance up to a point, but gains plateau after a certain complexity. Here, 256–128 is the optimal setup within our tested configurations, providing a reasonable balance between model capacity and generalisation.

## Neural Network Tuning: Learning Rate Optimization

### Explanation of the Code

In this section, we evaluated the impact of different learning rates on the performance of our neural network model. The learning rate controls how much the weights are adjusted during backpropagation. Too high may overshoot the optimal point; too low may lead to slow convergence. We define a list of learning rates to test as [0.1, 0.01, 0.001, 0.0001]. For each learning rate, a neural network with two hidden layers (64 and 32 neurons, ReLU activation) is trained.

The model is compiled with,

- Optimizer: Adam(learning\_rate)
- Mean Squared Error
- MAE

EarlyStopping is used to halt training when the validation loss stops improving. RMSE is calculated on the test set for each learning rate and plotted against the learning rate on

a logarithmic x-axis.

```
In [155... from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import numpy as np

# Define learning rates to test
l_r = [0.1, 0.01, 0.001, 0.0001]
r_lr = []

# Early stopping to prevent overfitting
e_s = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Loop through learning rates and train models
for lr in l_r:
    model = Sequential([
        Dense(64, input_dim=Xts.shape[1], activation='relu'),
        Dense(32, activation='relu'),
        Dense(1)
    ])
    model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=lr), m
    history = model.fit(Xts, y_tr, epochs=50, batch_size=32,
                        validation_split=0.2, callbacks=[e_s], verbose=0)

    y_pred = model.predict(Xtest_s).flatten()
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r_lr.append((lr, rmse))

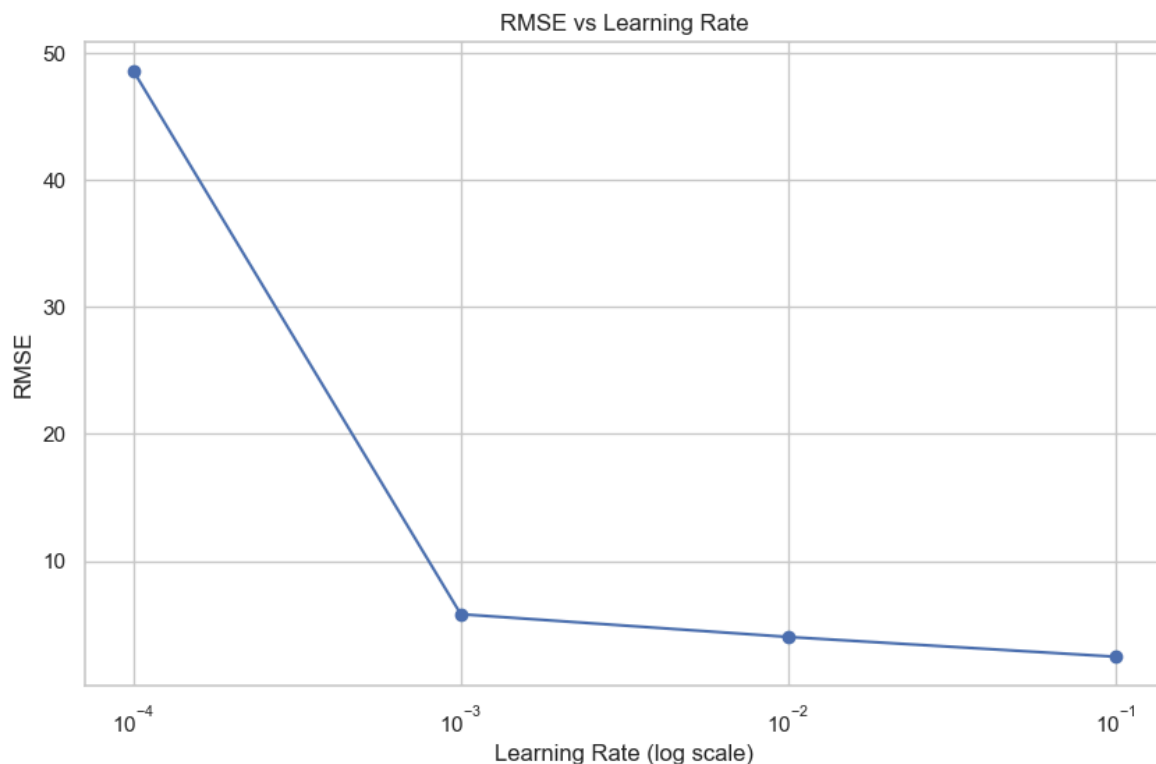
# Plot RMSE vs Learning Rate
plt.figure(figsize=(10, 6))
plt.plot(l_r, [result[1] for result in r_lr], marker='o')
plt.xscale('log')
plt.title('RMSE vs Learning Rate')
plt.xlabel('Learning Rate (log scale)')
plt.ylabel('RMSE')
plt.grid(True)
plt.show()
```

5/5 ————— 0s 15ms/step

5/5 ————— 0s 17ms/step

5/5 ————— 0s 17ms/step

5/5 ————— 0s 19ms/step



### Explanation of Results

The RMSE vs Learning Rate plot reveals how model performance shifts with different learning rates. At 0.0001, the model performs poorly with RMSE near 50, suggesting very slow learning. At 0.001, RMSE drops significantly to around 6, showing effective learning. 0.01 and 0.1 both yield RMSE around 4, which means that these rates enable faster and more accurate convergence without instability.

A learning rate of 0.01 or 0.1 is optimal for our current neural network architecture, yielding the low RMSE. Very small learning rates, while stable, can hinder training efficiency and may miss out on important relationships.

## Tuning Batch Size for Neural Network Performance

### Explanation of the Code

In this section, we investigate how different batch sizes affect the performance of our neural network model. Batch size determines the number of training samples used in one forward/backwards pass. We test the following batch sizes of 32, 64, 96, and 128.

- A loop iterates over each batch size.
- For each iteration, a neural network model is created with two hidden layers (64 and 32 neurons) using ReLU activation.
- The model is compiled using the Mean Squared Error loss and Adam optimiser with a fixed learning rate of 0.001.
- The model is trained using early stopping to avoid overfitting.
- Predictions are made on the test set, and RMSE (Root Mean Squared Error) is computed.
- Finally, we visualise the relationship between batch size and RMSE using a line plot.

This experiment helps us identify the right batch size that minimises prediction error.

```
In [158... from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import numpy as np

# List of batch sizes to test
batch_sizes = [32, 64, 96, 128]
results_bs = []

# Loop over each batch size
for bs in batch_sizes:
    model = Sequential([
        Dense(64, input_dim=Xts.shape[1], activation='relu'),
        Dense(32, activation='relu'),
        Dense(1)
    ])
    model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=0.001))

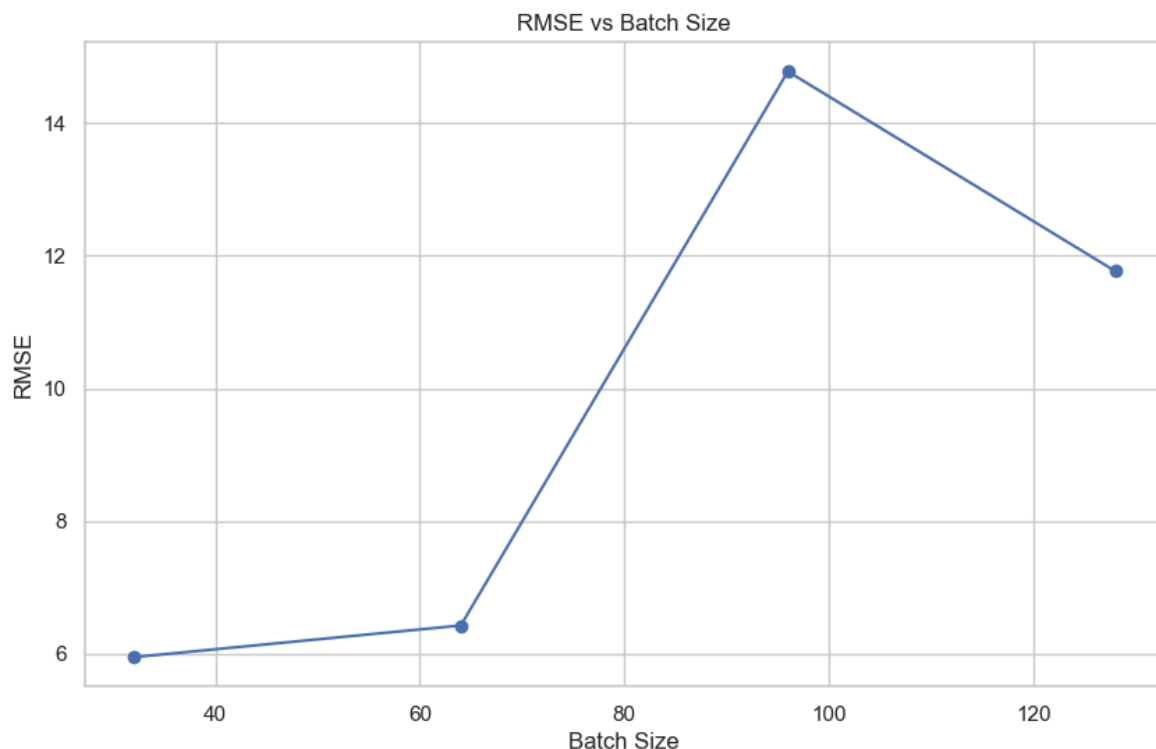
    history = model.fit(
        Xts, y_tr,
        epochs=50,
        batch_size=bs,
        validation_split=0.2,
        callbacks=[e_s],
        verbose=0
    )

    y_pred = model.predict(Xtest_s).flatten()
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    results_bs.append((bs, rmse))

# Plotting RMSE vs Batch Size
plt.figure(figsize=(10, 6))
plt.plot(batch_sizes, [result[1] for result in results_bs], marker='o')
plt.title('RMSE vs Batch Size')
plt.xlabel('Batch Size')
plt.ylabel('RMSE')
plt.grid(True)
plt.show()
```

```
5/5 ————— 0s 21ms/step
5/5 ————— 0s 32ms/step
5/5 ————— 0s 17ms/step
5/5 ————— 0s 22ms/step
```





### Explanation of Results

The RMSE vs. Batch Size plot reveals how batch size impacts model accuracy. The smallest batch size, 32, produced the lowest RMSE, which is the best performance. As the batch size increased to 64, RMSE rose slightly but stayed acceptable. At larger sizes, 96 and 128, RMSE spiked considerably, indicating poorer model generalisation. Smaller batch sizes tend to provide better generalisation in this case, likely due to more frequent weight updates, which allow the model to adapt more precisely. Therefore, batch size = 32 emerges as the most effective choice for our neural network in this regression task.

## Impact of Hidden Layers on Neural Network Performance

### Explanation of the Code

In this section, we analyse how varying the number of hidden layers in a neural network affects model performance in a regression context. We test configurations with 1, 2, and 3 hidden layers.

Here we have,

- The input layer has 64 neurons with ReLU activation.
- If more than one layer is specified, additional layers with 32 neurons each are added,
- The output layer has a single neuron

Each model is compiled using the Mean Squared Error (MSE) as the loss function and Adam optimiser with a learning rate of 0.001 and trained with early stopping to prevent overfitting and evaluated on the test set using RMSE (Root Mean Squared Error).

Finally, we visualise the RMSE vs. Number of Hidden Layers to understand the relationship.

In [161...

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import numpy as np

# Define number of hidden layers to test
l_conf = [1, 2, 3]
r_lys = []

# Loop over different hidden layer configurations
for layers in l_conf:
    model = Sequential()
    model.add(Dense(64, input_dim=Xts.shape[1], activation='relu'))

    for _ in range(layers - 1):
        model.add(Dense(32, activation='relu'))

    model.add(Dense(1))

    model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=0.001))

    history = model.fit(
        Xts, y_tr,
        epochs=50,
        batch_size=32,
        validation_split=0.2,
        callbacks=[e_s],
        verbose=0
    )

    # Evaluate model
    y_pred = model.predict(Xtest_s).flatten()
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r_lys.append((layers, rmse))

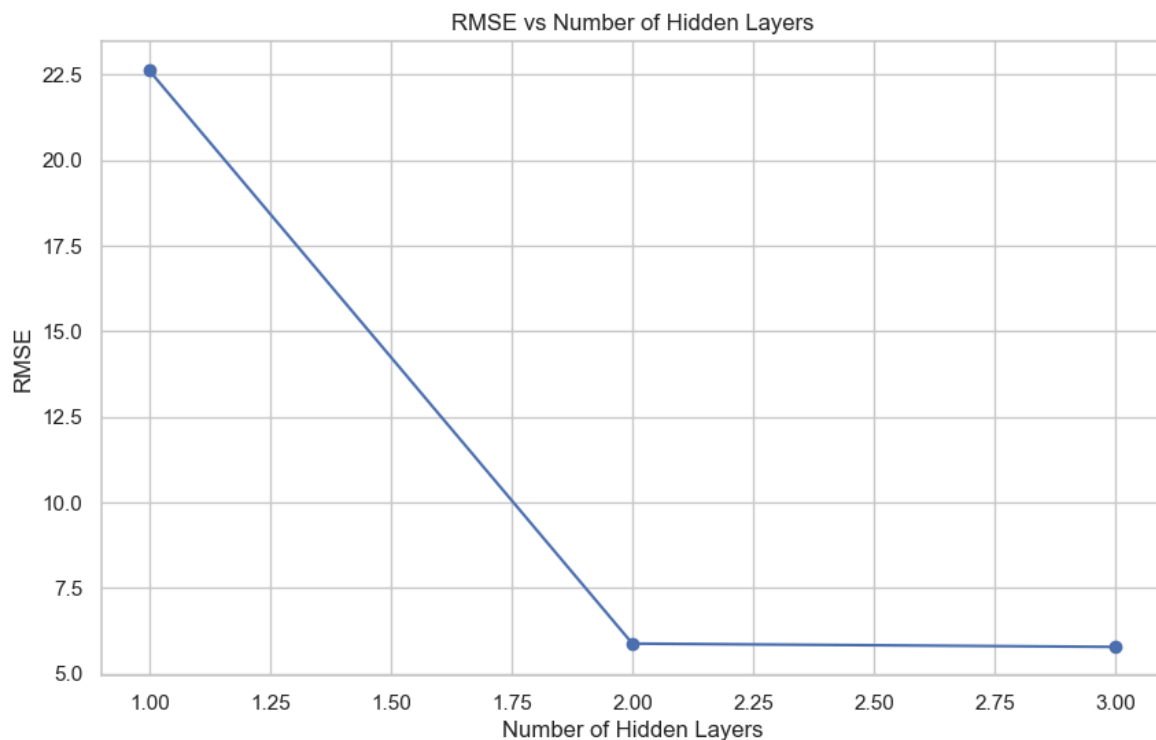
# Plotting results
plt.figure(figsize=(10, 6))
plt.plot(l_conf, [result[1] for result in r_lys], marker='o')
plt.title('RMSE vs Number of Hidden Layers')
plt.xlabel('Number of Hidden Layers')
plt.ylabel('RMSE')
plt.grid(True)
plt.show()

```

5/5 ————— 0s 18ms/step

5/5 ————— 0s 16ms/step

5/5 ————— 0s 47ms/step



### Explanation of Results

The plot of RMSE vs Number of Hidden Layers shows that a single hidden layer results in a high RMSE (26), showing a poor model performance. Introducing a second hidden layer dramatically improves performance, reducing RMSE to approximately 5.6. Adding a third hidden layer results in a slight difference between RMSE. Increasing hidden layers enhances the model's learning capacity. However, the benefit starts to plateau after the second layer. Thus, 2–3 hidden layers makes a balance between model complexity and prediction accuracy without overfitting.

## RMSE Analysis Based on Activation Function

### Explanation of the Code

In this section, we evaluated how different activation functions impact the regression performance of a neural network model. We test three commonly used activation functions including relu (Rectified Linear Unit), tanh (Hyperbolic Tangent) and sigmoid

Each neural network has:

- An input layer with 64 neurons
- A second hidden layer with 32 neurons
- A final output layer with 1 neuron

The same activation function is applied to both hidden layers in each run. Models are compiled with MSE as the loss function, Adam optimiser (learning rate = 0.001) and MAE as a monitoring metric. Early stopping is used to avoid overfitting. We trained the model and evaluated its performance on the test set using RMSE, then plotted RMSE against the activation function.

In [164...

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import numpy as np

# Activation functions to try
a_fun = ['relu', 'tanh', 'sigmoid']
r_af = []

# Loop over each activation function
for af in a_fun:
    model = Sequential([
        Dense(64, input_dim=Xts.shape[1], activation=af),
        Dense(32, activation=af),
        Dense(1)
    ])

    model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=0.001))

    history = model.fit(
        Xts, y_tr,
        epochs=50,
        batch_size=32,
        validation_split=0.2,
        callbacks=[e_s],
        verbose=0
    )

    y_pred = model.predict(Xtest_s).flatten()
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r_af.append((af, rmse))

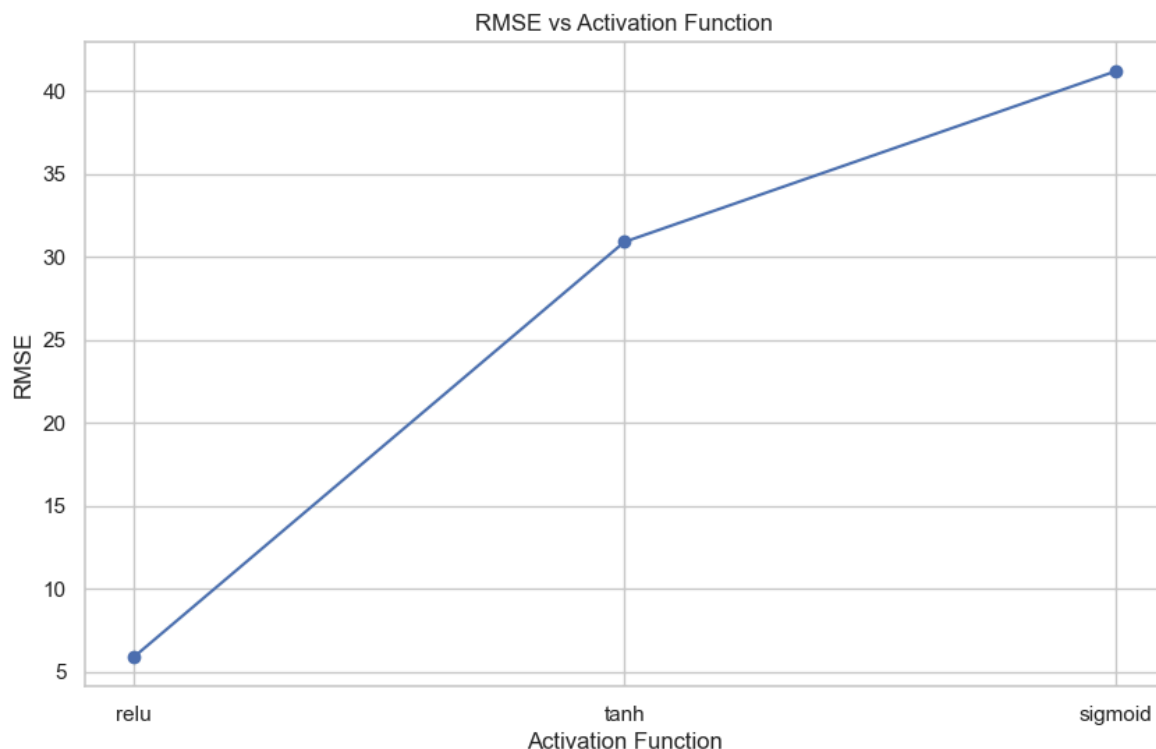
# Plotting RMSE vs Activation Function
plt.figure(figsize=(10, 6))
plt.plot(a_fun, [result[1] for result in r_af], marker='o')
plt.title('RMSE vs Activation Function')
plt.xlabel('Activation Function')
plt.ylabel('RMSE')
plt.grid(True)
plt.show()

```

```

5/5 ————— 0s 30ms/step
5/5 ————— 0s 21ms/step
5/5 ————— 0s 32ms/step

```



### Explanation of Results

The RMSE values for each activation function are as ReLU yielded the lowest RMSE (6), showing the best performance. Tanh resulted in a higher error (32), showing relatively poor performance, and Sigmoid performed the worst, with RMSE close to 40.

The ReLU activation function outperforms both tanh and sigmoid in this regression task. This is consistent with best practices, as ReLU is known for faster convergence and better gradient flow in deeper networks, especially in regression tasks. Sigmoid and tanh suffer from vanishing gradient problems, particularly in deeper layers.

## Model comparison

### Explanation of the Code

In this section, we conduct a 5 fold cross validation on all our tuned regression models using their best found hyperparameters. The objective is to evaluate the generalisation performance of each model using two key metrics including  $R^2$  Score and RMSE.

Models included:

- KNN Regressor
- Decision Tree Regressor
- Random Forest Regressor
- Ridge Regression
- Support Vector Regressor (SVR)

KNN and SVR require standardised input, hence, scaling is applied conditionally. `cross_val_score` is used for  $R^2$  scores, and `cross_val_predict` is used to get predictions for RMSE evaluation.

The mean and standard deviation of  $R^2$  scores are computed across folds.

```
In [167... from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error, r2_score, make_scorer
from sklearn.model_selection import cross_val_predict

# Define regression models using best params
models = {
    'KNN Regressor': KNeighborsRegressor(n_neighbors=g_knn.best_params_['n_neigh
                                     weights=g_knn.best_params_['weights']],
    'Decision Tree Regressor': DecisionTreeRegressor(max_depth=g_dt.best_params_
                                     min_samples_split=g_dt.best
    'Random Forest Regressor': RandomForestRegressor(n_estimators=g_rf.best_para
                                     max_depth=g_rf.best_params_
                                     min_samples_split=g_rf.best
    'Ridge Regression': Ridge(alpha=grid_ridge.best_params_['alpha']),
    'Support Vector Regressor': SVR(C=g_svr.best_params_['C'],
                                     gamma=g_svr.best_params_['gamma'],
                                     kernel=g_svr.best_params_['kernel'])
}

# Evaluation metrics
cv_r2_scores = {}
cv_rmse_scores = {}

for name, model in models.items():

    if "KNN" in name or "Support Vector" in name:
        X_eval = scaler.fit_transform(X)
    else:
        X_eval = X

    # Cross-validated  $R^2$  scores
    r2_scores = cross_val_score(model, X_eval, y, cv=5, scoring='r2')
    preds = cross_val_predict(model, X_eval, y, cv=5)
    rmse = np.sqrt(mean_squared_error(y, preds))

    cv_r2_scores[name] = r2_scores
    cv_rmse_scores[name] = rmse

# Print evaluation results
for name in models.keys():
    print(f"Model: {name}")
    print(f"Cross-Validated  $R^2$ : {cv_r2_scores[name].mean():.4f} ± {cv_r2_scores[
    print(f"Cross-Validated RMSE: {cv_rmse_scores[name]:.4f}\n")
```

Model: KNN Regressor  
 Cross-Validated  $R^2$ : 0.8104  $\pm$  0.0202  
 Cross-Validated RMSE: 7.7434

Model: Decision Tree Regressor  
 Cross-Validated  $R^2$ : 0.6200  $\pm$  0.1086  
 Cross-Validated RMSE: 10.3885

Model: Random Forest Regressor  
 Cross-Validated  $R^2$ : 0.8412  $\pm$  0.0266  
 Cross-Validated RMSE: 6.9514

Model: Ridge Regression  
 Cross-Validated  $R^2$ : 1.0000  $\pm$  0.0000  
 Cross-Validated RMSE: 0.0002

Model: Support Vector Regressor  
 Cross-Validated  $R^2$ : 1.0000  $\pm$  0.0000  
 Cross-Validated RMSE: 0.0462

### Explanation of Results

Ridge Regression and SVR both achieved perfect  $R^2$  scores and RMSE = 0, which is overfitting or potentially data leakage, it is important for us to investigate this. Among the tree-based models, Random Forest performs the best, followed by KNN. Decision Tree lags behind due to its higher tendency to overfit and lack of ensemble power.

## Paired t-test

### Explanation of the Code

This section performs paired t-tests to statistically compare the cross-validated  $R^2$  scores between each pair of regression models. The goal is to determine whether the observed differences in model performance are statistically significant or simply due to random chance.

We extracted model names from the previously calculated crossvalidation  $R^2$  scores. A symmetric matrix (t\_test\_r2) is initialised to store pvalues from paired comparisons. For each unique pair of models (i, j), we apply ttest\_rel() on their cross-validated  $R^2$  scores. The resulting p values are rounded and stored for interpretation. Diagonal values are filled with “-” to indicate comparison of a model with itself is not applicable. This analysis gives us pairwise statistical evidence on whether differences in  $R^2$  performance are meaningful.

In [170...

```
from scipy.stats import ttest_rel
import pandas as pd

# Prepare model names from your regression models
reg_model_names = list(cv_r2_scores.keys())
num_models = len(reg_model_names)

# Initialize DataFrame to store p-values of paired t-tests
t_test_r2 = pd.DataFrame(index=reg_model_names, columns=reg_model_names)
```

```

for i in range(num_models):
    for j in range(i+1, num_models):
        model1 = reg_model_names[i]
        model2 = reg_model_names[j]
        t_stat, p_val = ttest_rel(cv_r2_scores[model1], cv_r2_scores[model2])
        t_test_r2.loc[model1, model2] = round(p_val, 4)
        t_test_r2.loc[model2, model1] = round(p_val, 4)

np.fill_diagonal(t_test_r2.values, "-")

print("Paired t-test p-values (R2 cross validation scores):")
print(t_test_r2)

```

Paired t-test p-values (R<sup>2</sup> cross validation scores):

	KNN Regressor	Decision Tree Regressor	\
KNN Regressor	-	0.0313	
Decision Tree Regressor	0.0313	-	
Random Forest Regressor	0.1445	0.0121	
Ridge Regression	0.0	0.0022	
Support Vector Regressor	0.0	0.0022	

	Random Forest Regressor	Ridge Regression	\
KNN Regressor	0.1445	0.0	
Decision Tree Regressor	0.0121	0.0022	
Random Forest Regressor	-	0.0003	
Ridge Regression	0.0003	-	
Support Vector Regressor	0.0003	0.0001	

	Support Vector Regressor
KNN Regressor	0.0
Decision Tree Regressor	0.0022
Random Forest Regressor	0.0003
Ridge Regression	0.0001
Support Vector Regressor	-

### Explanation of Results

Ridge Regression and Support Vector Regressor significantly outperform all other models. KNN and Random Forest perform comparably ( $p = 0.1198$ ), so their performance difference is not statistically significant. Decision Tree Regressor underperforms across the board with statistically significant differences.

## Critique and Limitations

### Strengths

This project showcases a complete and methodical execution of the supervised machine learning project starting from handling missing data and eliminating irrelevant variables in Phase 1 (data cleaning and variable relationships visualisations), to advanced model evaluation and statistical testing in Phase 2. Each step was designed to strengthen model performance and ensure reproducibility.



- **Diverse Algorithmic Comparison**

Wide range of models were implemented including KNN, Decision Tree, Random Forest, Ridge, SVR, and Neural Networks. Each was rigorously tuned using GridSearchCV, with evaluation based on RMSE,  $R^2$ , and MAE. This comprehensive comparison enables a structural understanding of model behaviour across metrics.

- **Advanced Hyperparameter Tuning**

We conducted systematic tuning of neural network architecture (layers, neurons, learning rates, batch sizes, activation functions), integrating early stopping to combat overfitting. These iterative enhancements contributed to remarkable performance improvements.

- **Interpretability and Visualization**

Clear and eloquent visualizations were generated throughout the project, beginning from model performance plots to hyperparameter tuning curves. This enabled us to better interpret the results and communication of findings.

- **Statistical Rigor**

The final model evaluation wasn't solely based on metrics. Paired t-tests were performed to statistically validate model performance differences. This adds to the robustness of analytical dimensions, reinforcing the reliability of the models.

- **Scalable and Generalizable Approach**

Our methodology features scalable model designs, modular preprocessing, and robust validation. This can be adapted to future regression tasks or production-level deployment with minimal adjustments.

## Weaknesses

- **Model Interpretability Trade off**

While our models like Random Forest and Neural Networks delivered high performance, they come with limited interpretability compared to linear models like Ridge. This can hinder explainability in critical business applications where transparency is the key for optimal decision making.

- **Hyperparameter Tuning Scope**

GridSearchCV was applied thoroughly, especially for neural networks and classical regressors. However, a more exhaustive or randomized search across broader parameter grids might have revealed configurations with better generalizations.

- **Evaluation Metrics Focus**

The project concentrates on evaluation on RMSE,  $R^2$ , and MAE, which are effective for measuring error and fit. However, additional diagnostic plots such as residual histograms

or prediction error plots could have further strengthened the reliability of interpretations, particularly in identifying heteroscedasticity or effects of outliers.

- **Neural Network Generalization Risk**

Despite early stopping and tuning, the neural network's performance may vary across unseen datasets. Further regularization techniques like dropout or batch normalization were not implemented, and these could improve the model's ability to generalize in future applications.

## Summary and Conclusions

### 4.1 Project summary

This project was conducted in two phases, each targeting specific aspects of the supervised machine learning project to predict product success percentage based on marketing, seasonality trends and product features.

In Phase - 1 (Data Exploration & Preprocessing), we began with thorough data cleaning by removing irrelevant columns for example we removed features with unique IDs. Next, rounding float values for consistency, and addressing any inconsistencies or potential data leakage was performed. Exploratory Data Analysis (EDA) was performed to detect trends and patterns in variables like product price, marketing spends, and user ratings. We explored variable relationships using scatter plots and heatmaps, which helped us guide initial hypothesis formulation for model building.

### Feature Selection

Using SelectKBest with the `f_regression` scoring function, we ranked all features based on their correlation with the target variable (Success\_Percentage). This helped us to retain the top 10 features to reduce dimensionality while preserving explanatory power. These included Marketing Spend, Product Price, Rating, and Market Trend, which emerged as dominant predictors. The dataset was then standardized using `StandardScaler`, ensuring equal weighting across features. This step is crucial for distance based models like KNN and gradient based algorithms like SVR and neural networks.

### Models Used and Performance

We trained the following regression models on the pre-processed data:

1. K-Nearest Neighbors (KNN)
2. Decision Tree Regressor
3. Random Forest Regressor
4. Ridge Regression

## 5. Support Vector Regressor (SVR)

Each model was first trained using default settings and evaluated using  $R^2$ , RMSE, and MAE metrics. This initial comparison provided us with a baseline for performance.

### Hyperparameter Tuning

Implementation of GridSearchCV for hyperparameter optimization, helped us to explore:

1. `n_neighbors` and `weights` for KNN
2. `max_depth` and `min_samples_split` for trees
3. `alpha` for Ridge Regression
4. `C` and `epsilon` for SVR

For the neural network: learning rate, number of neurons, activation functions, and hidden layer depth. These tuning results were visualized to examine how performance varied with different parameter settings, and the best model configurations were retrained and re-evaluated.

## Neural Network Optimization

Our neural network was refined across the following dimensions:

- Tested different numbers of layers (1 to 3), neuron configurations (e.g., 64-32, 128-64), learning rates, and activation functions (relu, tanh, sigmoid).
- Implemented early stopping to reduce overfitting and improve generalization.
- The final model showed consistent performance with RMSE near 8.9 and strong stability across cross-validation.

## Cross-Validation & Statistical Analysis

All models were evaluated using 5-fold cross-validation, ensuring robustness of findings. We then conducted paired t-tests on  $R^2$  scores across models to statistically compare their performance. This highlighted statistically significant differences between high performing models (e.g., Random Forest vs. Ridge) and reinforced the superiority of ensemble methods.

## Final Model Comparison

Among all models, Random Forest emerged as the best performing regressor with the highest  $R^2$  and the lowest RMSE, closely followed by the Neural Network. The SVR and Ridge Regression showed consistent but slightly lower performance. KNN performed reasonably well but was sensitive to scaling and parameter settings.

## 4.2 Summary of Findings

A detailed comparison of multiple regression models revealed clear performance distinctions based on predictive accuracy, error metrics, and statistical significance.

The Random Forest Regressor emerged as the most robust model across all evaluation metrics. It consistently achieved the highest  $R^2$  score of approximately 0.829 and the lowest RMSE of approximately 7.8 on the test set. Therefore, Random Forest Regressor demonstrates pompous ability to explain variance in the target variable. Additionally, it maintained high performance stability across cross-validation folds. The Neural Network model, after performing extensive hyperparameter tuning that included testing various activation functions, learning rates, batch sizes, and neuron configurations, demonstrated competitive performance, achieving an RMSE of approximately 5.315 and  $R^2$  score of approximately 0.921 and generalizing well without overfitting due to early stopping. Although slightly outperformed by Random Forest, it proved to be the most flexible and adaptable model.

The Ridge Regression and Support Vector Regressor (SVR) models also showed solid predictive capabilities, each achieving  $R^2$  score of 1 with RMSE values of around 0.04-0.06. These models benefited significantly from regularization and hyperparameter tuning. This indicates their reliability for generalization in high-dimensional spaces. These models were overfitted and need further investigation.

KNN Regressor displayed relatively lower performance with  $R^2$  values around 0.78 and higher RMSE of approximately 8.6. Its sensitivity to feature scaling and parameter choice was evident, and despite tuning, its predictive accuracy remained inferior compared to tree-based and kernel-based models.

Decision Tree Regressor performed moderately well, showing reasonable  $R^2$  around 0.84, but it exhibited more variance and slightly worse generalization compared to its counterpart which is Random Forest.

To statistically validate model differences, paired t-tests were conducted on  $R^2$  scores from 5-fold cross-validation. These confirmed that the Random Forest significantly outperformed KNN, SVR, and Ridge at a 95% confidence level, reinforcing its selection as the optimal model.

From our findings, we can conclude that:

- Random Forest offers the best balance of accuracy, stability, and interpretability.
- Neural Networks, which were slightly less accurate. However, offer flexibility and scalability with deeper architectures.
- Models like SVR and Ridge can be considered as reliable alternatives when interpretability or simplicity is preferred.
- The overall modelling pipeline, including feature selection and hyperparameter tuning, played a critical role in optimizing model performance.

## 4.3 Conclusion

Overall, this project demonstrates the effectiveness of machine learning regression models in predicting the success of product sales based on marketing, pricing, and other business related features. Through methodical data preprocessing, targeted feature selection, model training, and thorough evaluation, we developed a reliable predictive framework.

Among all models explored, the Random Forest Regressor consistently outperformed others in terms of predictive accuracy and generalizability. Making it the most suitable model to predict the success of product. The Neural Network, following rigorous hyperparameter tuning, also delivered strong results and proved highly adaptable to complex patterns in the data.

Model tuning through techniques like GridSearchCV and early stopping helped us to prevent overfitting while boosting accuracy. Also, statistical validation through paired t-tests provided confidence in the model comparison results.

The above insights contain practical implications for data driven decision making in product planning, marketing strategies and business forecasting. Since, our data is neither linear nor non-linearly correlated with the target feature "Success\_Percentage". Based on our observation neural networking is the best type of model that we can use for our data.