132C Final Project

Team Member: Yulun Wu, Chang Yuan, Renyu Wang

March 2024

1 Introduction

As the most commonly consumed fruit, apple plays an important role in our daily nutrition supplement. According to the United States Department of Agriculture, 15.77 pounds of fresh apples were consumed per American consumer in 2022 and 2023.[1] As consumers, we always want to buy sweet and fresh apples from groceries. However, determining the sweetness of apples by touching and observing is not very accurate. As a result, in this project, we implement the Support Vector Regression (SVR) model and Decision Tree model to determine the factors that affect the sweetness of apples and statistically predict the sweetness of any given apple.

2 Feature Selection

Before choosing the appropriate models, we first want to determine what factors can affect the sweetness of apples. By reading several papers, we find that hardness, size, and ripeness may be highly correlated to sweetness.[2] Then we find an online data set including size, weight, sweetness, crunchiness, juiciness, ripeness, and acidity from Kaggle[3] and perform a correlation test by calculating Pearson and Spearman correlation coefficients relative to Sweetness in order to decide the features we will use in our models. The result is shown below. The number in the figure is Pearson and Spearman correlation coefficients. The closer the number is to

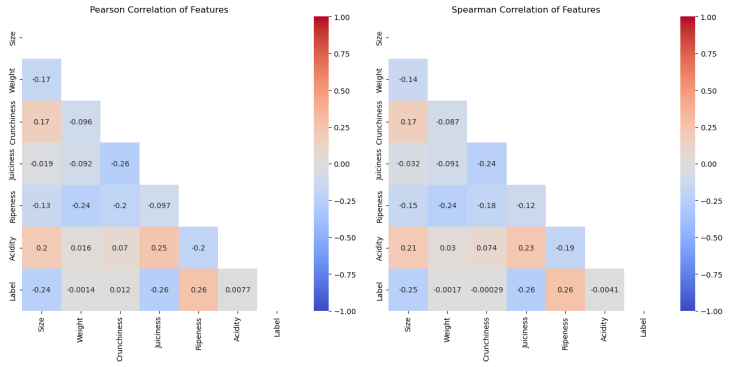

Figure 1: Correlation Heat Map

zero, the less relevant to sweetness. The negative number here means a negative correlation. However, the fact that both Pearson and Spearman correlations are relatively low for most feature pairs implies that we have complex relationships between features rather than simple

linear and monotonic associations, which means that we need to choose models that are capable for complex correlations. Finally, we choose ripeness, size, and weight as our main target features due to their relatively large correlation coefficients in both figures.

3 Model Selection

To pre-process datasets, we scaled it using Z-score normalization to ensure consistent feature scales. We then split the dataset into training (70%) and testing (30%) sets and employed k-fold cross-validation to mitigate overfitting risks. To identify the most appropriate models, we calculated the Root Mean Squared Error (RMSE) for six potential candidates: Random Forest, Decision Tree, Linear Regression, Gradient Boosting, K-Nearest Neighbors, and Support Vector Regression (SVR). All these models are adept at capturing complex correlations, with lower RMSE values around 1 indicating better performance. Ultimately, we narrowed our focus to the SVR and Decision Tree models due to their respective smallest and largest RMSE scores. This contrast induced our curiosity about how the difference in RMSE would translate to prediction accuracy and reliability.

4 SVR Model

The Support Vector Regression (SVR) model outperformed the Decision Tree Regressor with a lower RMSE of 1.0516, demonstrating greater accuracy in predicting apple sweetness. This precision reflects SVR's effectiveness in managing the dataset's complexities and nonlinear relationships. While requiring careful hyperparameter tuning to reach its full potential, the SVR model's ability to provide accurate and reliable predictions without overfitting makes it a superior choice for this application, balancing complexity with predictive power.

5 Decision Tree Model

While the Decision Tree Regressor offered intuitive interpretability for understanding how features like juiciness and acidity influence sweetness, it achieved a higher RMSE of 1.6232, indicating less precise predictions compared to the SVR model. However, the Decision Tree model exhibited overfitting tendencies, a common limitation due to capturing noise in training data. Despite its interpretability strength, careful complexity control is needed to mitigate overfitting issues.

6 Model Comparison

The simulation result from the two models is shown below. It is obvious that the SVR model shows a more converging result relative to actual sweetness. The Support Vector Regression (SVR) model demonstrated superior accuracy with a lower RMSE in predicting apple sweetness, showcasing its robustness in handling complex relationships. Although the Decision Tree Regressor offered straightforward interpretability for understanding feature influences, it suffered from a higher RMSE and overfitting tendencies. Prioritizing predictive accuracy led

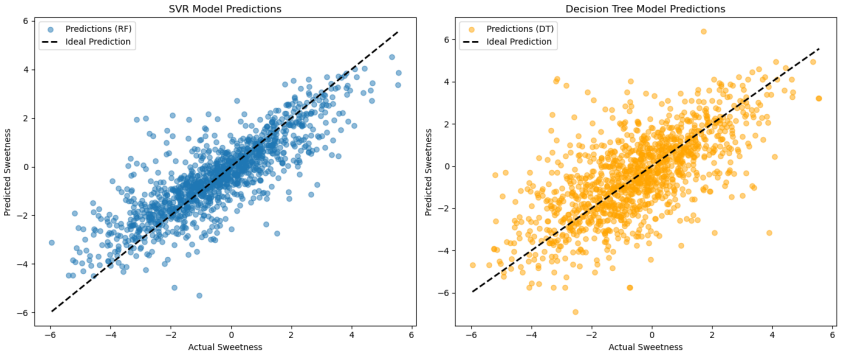

Figure 2: Predictions from SVR and Decision Tree Models

to the selection of the SVR model over the more interpretable but less precise Decision Tree approach. Our next step is to perform Hyperparameter Tuning for the SVR Model to improve its performance

7 Delve into SVR Model

Support Vector Regression (SVR) extends SVMs to regression by fitting the best line/hyperplane within a tolerance ϵ. Using the ϵ-insensitive loss minimizes errors exceeding ϵ for robustness without overfitting. The kernel trick allows SVR to handle complex, high-dimensional data, making it valuable across fields like finance, energy, climate, and healthcare for solving diverse regression problems.

Our project significantly improved SVR performance through meticulous hyperparameter tuning of regularization (C), kernel, gamma (γ), and epsilon (ϵ) to find the optimal combination minimizing prediction error. Pre-tuning, SVR showed solid predictive capabilities, but post-tuning accuracy enhanced, reducing RMSE. This underscores tuning's importance in reducing overfitting, increasing generalizability, and optimizing SVR's predictive power. The improved SVR model predictions are shown in Appendix A, Figure 3. We observed that the results are more converging than the previous results, which means that the new parameters did improve the performance of the SVR model. Finally, we calculated the permutation importance for each feature and the result is shown in Figure 4 Appendix A. Features with outstanding importance correspond to the results of the correlation analysis, which means that you should consider these characteristics to pick the sweetest apple.

8 Conclusion

Our prediction model holds significant practical implications for both agriculture and consumer markets. For farmers, it enables optimizing harvest timing and guiding selective breeding efforts to cultivate sweeter apple varieties. In the consumer space, fruit companies can leverage these insights to tailor new apple-based products that align with consumer preferences for sweetness. The model's broad utility in improving product quality and market efficiency underscores its value as a pivotal advancement in leveraging data-driven insights to optimize fruit quality and marketability.

A Supplement Figure

Figure 3: Best SVR Predictions

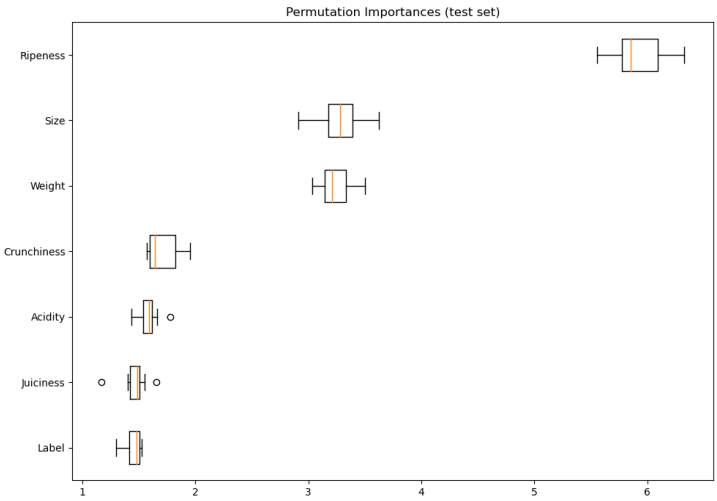

Figure 4: Permutation Importance for Features

B Code

```
# 132C Final Project Apple Sweetness Predictive Model
# Group Member: Yulun Wu, Max Wang, Chang Yuan
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# Load the dataset
file_path = "C://Users//Lun//Desktop//apple_quality.csv"
df = pd.read_csv(file_path)

# Display the first 5 rows of the dataset
print(df.head())

# Converting the 'Acidity' column to numeric
df['Acidity'] = pd.to_numeric(df['Acidity'], errors='coerce')


def clean_data(df):

    df = df.drop(columns=['A_id'])

    df = df.dropna()

    df = df.astype({'Acidity': 'float64'})

    def label(Quality):
        """
        Transform based on the following examples:
        Quality    Output
        "good"  => 0
        "bad"   => 1
        """
        if Quality == "good":
            return 0

        if Quality == "bad":
            return 1

        return None

    df['Label'] = df['Quality'].apply(label)
```

```
    df = df.drop(columns=['Quality'])

    df = df.astype({'Label': 'int64'})

    return df

df_clean = clean_data(df.copy())
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Calculate Pearson and Spearman correlation coefficients relative to
↪  Sweetness
pearson_correlation =
↪  df_clean.corr(method='pearson')[["Sweetness"]].sort_values(by="Sweetness",
↪  ascending=False)
spearman_correlation =
↪  df_clean.corr(method='spearman')[["Sweetness"]].sort_values(by="Sweetness",
↪  ascending=False)

# Combine both correlation coefficients into a single DataFrame for comparison
correlation_comparison_df = pd.DataFrame({
    'Pearson Correlation': pearson_correlation["Sweetness"],
    'Spearman Correlation': spearman_correlation["Sweetness"]
})

print(correlation_comparison_df)

# Drop 'Sweetness' column to avoid comparing it with itself
df_features = df_clean.drop('Sweetness', axis=1)

# Calculate Pearson and Spearman correlation matrices
pearson_corr = df_features.corr(method='pearson')
spearman_corr = df_features.corr(method='spearman')

# Create a mask to hide the upper triangle and the diagonal
mask = np.triu(np.ones_like(pearson_corr, dtype=bool))

# Plotting the Pearson Correlation Heatmap without the diagonal and upper
↪  triangle
plt.figure(figsize=(14, 7))
```

```
plt.subplot(1, 2, 1)
sns.heatmap(pearson_corr, mask=mask, annot=True, cmap='coolwarm', vmin=-1,
↪  vmax=1)
plt.title('Pearson Correlation of Features')

# Plotting the Spearman Correlation Heatmap without the diagonal and upper
↪  triangle
plt.subplot(1, 2, 2)
sns.heatmap(spearman_corr, mask=mask, annot=True, cmap='coolwarm', vmin=-1,
↪  vmax=1)
plt.title('Spearman Correlation of Features')

plt.tight_layout()
plt.show()

# Create a correlation heatmap
corr = df_clean.corr()

mask = np.triu(np.ones_like(corr, dtype=bool))

plt.figure(figsize=(10, 6))

cmap = sns.diverging_palette(230, 20, as_cmap=True)

sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5}, annot=True)

plt.title('Correlation Heatmap')
plt.show()


# Split the dataset into training and testing sets
from sklearn.model_selection import train_test_split
# Assuming 'Sweetness' is your target variable and you want to predict its
↪  value
# Define features (X) by dropping the 'Sweetness' column, and 'Sweetness' as
↪  the target variable (y)
X = df_clean.drop('Sweetness', axis=1)
y = df_clean['Sweetness']

# Split the dataset into training (70%) and testing (30%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
↪  random_state=42)
```

```
# Now X_train and y_train are your training features and labels, respectively,
# and X_test and y_test are your testing features and labels.
# Training Models
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from sklearn.model_selection import cross_val_score, KFold
from sklearn.metrics import mean_squared_error
import numpy as np

# define a function to perform k-fold cross-validation and return the RMSE for
↪  each model
def evaluate_model(model, X, y, n_splits=10):
    # Initialize the KFold parameters
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)

    # Perform cross-validation,
    mse_scores = cross_val_score(model, X, y,
    ↪  scoring='neg_mean_squared_error', cv=kf)

    # Convert MSE scores to RMSE scores
    rmse_scores = np.sqrt(-mse_scores)

    # Return the average RMSE and its standard deviation
    return rmse_scores.mean(), rmse_scores.std()
# Initialize the models and evaluate relevant performance metrics
rf_model = RandomForestRegressor(random_state=42)
dt_model = DecisionTreeRegressor(random_state=42)
gb_model = GradientBoostingRegressor(random_state=42)
lr_model = LinearRegression()
knn_model = KNeighborsRegressor()
svr_model = SVR()

# Evaluate the Random Forest model
rf_rmse_mean, rf_rmse_std = evaluate_model(rf_model, X_train, y_train)
print(f"Random Forest RMSE: {rf_rmse_mean} (± {rf_rmse_std})")

# Evaluate the Decision Tree model
dt_rmse_mean, dt_rmse_std = evaluate_model(dt_model, X_train, y_train)
```

```
print(f"Decision Tree RMSE: {dt_rmse_mean} (± {dt_rmse_std})")

# Evaluate the Linear Regression model
lr_rmse_mean, lr_rmse_std = evaluate_model(lr_model, X_train, y_train)
print(f"Linear Regression RMSE: {lr_rmse_mean} (± {lr_rmse_std})")

# Evaluate the Gradient Boosting model
gb_rmse_mean, gb_rmse_std = evaluate_model(gb_model, X_train, y_train)
print(f"Gradient Boosting RMSE: {gb_rmse_mean} (± {gb_rmse_std})")

# Evaluate the K-Nearest Neighbors model
knn_rmse_mean, knn_rmse_std = evaluate_model(knn_model, X_train, y_train)
print(f"K-Nearest Neighbors RMSE: {knn_rmse_mean} (± {knn_rmse_std})")

# Evaluate the Support Vector Regression model
svr_rmse_mean, svr_rmse_std = evaluate_model(svr_model, X_train, y_train)
print(f"Support Vector Regression RMSE: {svr_rmse_mean} (± {svr_rmse_std})")
Random Forest RMSE: 1.1207396979125113 (± 0.05940036817570841)
Decision Tree RMSE: 1.6231949679416278 (± 0.0868686005192945)
Linear Regression RMSE: 1.4955384460919015 (± 0.038763725309871995)
Gradient Boosting RMSE: 1.2553431889930957 (± 0.05944559154634673)
K-Nearest Neighbors RMSE: 1.1490799927719406 (± 0.05779535201224092)
Support Vector Regression RMSE: 1.051636108967516 (± 0.05813804912527862)
# Random Forest Regressor shows a good balance between accuracy and
↪  variability with an RMSE of 1.1207 ± 0.0594.
    # It's among the better models but not the best.
# Decision Tree Regressor has the highest RMSE at 1.6232 ± 0.0869, indicating
↪  it might be overfitting to
    # the training data or not capturing the underlying patterns well.
# Linear Regression performs reasonably with an RMSE of 1.4955 ± 0.0388,
↪  suggesting that the relationship between
    # features and target might be linear to some extent but not perfectly so.
# Gradient Boosting Regressor shows an improved performance over the Decision
↪  Tree with an RMSE of 1.2553 ± 0.0594,
    # benefiting from the ensemble learning approach to reduce overfitting and
    ↪  enhance prediction.
# K-Nearest Neighbors (KNN) has a competitive RMSE of 1.1491 ± 0.0578,
↪  indicating that locality and similarity play
    # significant roles in predicting sweetness.
# Support Vector Regression (SVR) offers the best performance among the models
↪  with an RMSE of 1.0516 ± 0.0581,
    # suggesting it's highly effective in capturing the complexity of the
    ↪  dataset.
```

```
# Train and predict with both models
svr_model.fit(X_train, y_train)
dt_model.fit(X_train, y_train)

svr_predictions = svr_model.predict(X_test)
dt_predictions = dt_model.predict(X_test)

# Create DataFrames to store actual and predicted values
svr_results_df = pd.DataFrame({'Actual Sweetness': y_test, 'Predicted
↪  Sweetness (RF)': svr_predictions})
dt_results_df = pd.DataFrame({'Actual Sweetness': y_test, 'Predicted Sweetness
↪  (DT)': dt_predictions})

# Combine both predictions for easier comparison
combined_results_df = svr_results_df.join(dt_results_df['Predicted Sweetness
↪  (DT)'])

# Display the first few rows of the combined results DataFrame
print(combined_results_df.head())

# Plotting the scatter plot for both models along with the y=x line for
↪  comparison
plt.figure(figsize=(14, 6))

# Random Forest predictions
plt.subplot(1, 2, 1)
plt.scatter(combined_results_df['Actual Sweetness'],
↪  combined_results_df['Predicted Sweetness (RF)'], alpha=0.5,
↪  label='Predictions (RF)')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--',
↪  lw=2, label='Ideal Prediction')
plt.xlabel('Actual Sweetness')
plt.ylabel('Predicted Sweetness')
plt.title('SVR Model Predictions')
plt.legend()

# Decision Tree predictions
plt.subplot(1, 2, 2)
plt.scatter(combined_results_df['Actual Sweetness'],
↪  combined_results_df['Predicted Sweetness (DT)'], alpha=0.5,
↪  color='orange', label='Predictions (DT)')
```

```
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--',
↪  lw=2, label='Ideal Prediction')
plt.xlabel('Actual Sweetness')
plt.ylabel('Predicted Sweetness')
plt.title('Decision Tree Model Predictions')
plt.legend()

plt.tight_layout()
plt.show()

# Hyperparameter Tuning for SVR
from sklearn.model_selection import GridSearchCV, cross_val_score, KFold

# Define the SVR model and parameter grid to search
model = SVR()
param_grid = {
    'C': [0.1, 1, 10, 100],  # Example values, adjust based on your dataset
    'kernel': ['linear', 'rbf', 'poly'],
    'gamma': ['scale', 'auto'],
    'epsilon': [0.01, 0.1, 0.5]
}

# Setup GridSearchCV
grid_search = GridSearchCV(model, param_grid, cv=5,
↪  scoring='neg_mean_squared_error', verbose=2)
grid_search.fit(X_train, y_train)

# Best parameters and score
print("Best parameters:", grid_search.best_params_)
print("Best score (neg_mean_squared_error):", grid_search.best_score_)

# Train and evaluate the best model
best_model = grid_search.best_estimator_

# Cross-validation to check robustness
cv_scores = cross_val_score(best_model, X_train, y_train,
↪  cv=KFold(n_splits=10, shuffle=True, random_state=42),
↪  scoring='neg_mean_squared_error')
cv_rmse_scores = np.sqrt(-cv_scores)
print("After hyperparameter tuning:")
print("SVR w/ CV RMSE scores:", cv_rmse_scores)
print("Mean Cross-Validation RMSE:", cv_rmse_scores.mean())
print("Standard deviation of CV RMSE:", cv_rmse_scores.std())
```

```
# Best parameters: {'C': 100, 'epsilon': 0.1, 'gamma': 'scale', 'kernel':
↪  'rbf'}
# The set of parameters gives the best performance based on the negative mean
↪  squared error (MSE) criterion. The parameters indicate
    # a relatively high penalty for misclassification (C=100), moderate
    ↪  sensitivity to the training data (epsilon=0.1), and the use
    # of the radial basis functrion (RBF) kernel, which is comonly a good
    ↪  choice for a variety of problems.
# The best score of -0.83940 for the negativce MSE translates to an MSE of
↪  0.83940, which is a good performance. This is a measure
    # of the average squared difference between the estimated values and the
    ↪  actual value.
# Cross-validation and Robustness Check
    # CV RMSE scores: the RMSE scores across our 10 folds of cross-validation
    ↪  range from 0.8331 to 0.9773, with a mean of 0.9214
    # abd a standard deviation of 0.0454. This shows a fairly consistent
    ↪  performance across different splits of the training data,
    # indicating that the model is robust and not overfitting to a particular
    ↪  subset of the data. Moreover, the standard devaition
    # is relatively low, at 0.0454, suggesting that the model's performance is
    ↪  stable across different folds, which is a good sign
    # of the model's generalizability.
# Now, let's approxiamte feature importance and visualize it
from sklearn.inspection import permutation_importance
# This function will repeatedly shuffle each feature and compute the change in
↪  the model's performance. Features that cause a
    # significant drop in performance when shuffled are considered important
    ↪  because the model relies on them for accurate predictions.
# Assuming your SVR model is named svr_model and is already trained

result = permutation_importance(best_model, X_test, y_test, n_repeats=10,
↪  random_state=42, scoring='neg_mean_squared_error')

# Extracting importance mean and std
importances_mean = result.importances_mean
importances_std = result.importances_std

import matplotlib.pyplot as plt

# Sorting features by importance
sorted_idx = result.importances_mean.argsort()
```

```
plt.figure(figsize=(10, 7))
plt.boxplot(result.importances[sorted_idx].T, vert=False,
↪  labels=X_test.columns[sorted_idx])
plt.title("Permutation Importances (test set)")
plt.tight_layout()
plt.show()


# Choosing between permutationimportance and comparing coefficients:
# SVR and Decisiton Tree Models: Since we are working with SVR and Decision
↪  Tree models, comapring cofficiencts directly is
# not straightforward enough in all cases. SVR, especially with non-linear
↪  kernels, does not provide coefficients
# that easily translate to feature importance. Thus, we can use permutation
↪  importance instead. This technique is
# model-agnostic, which means it can be applied to any model type, including
↪  SVR. It provides a direct measure
# of impact on model performance whe a feature's value is randomly shuffled,
↪  offering a practical way to compare
# the importance of features across different models.
# Now, we are evaluating the tuned model and see how the performance has been
↪  enhanced
best_model_prediction = best_model.predict(X_test)
best_results_df = pd.DataFrame({'Actual Sweetness': y_test, 'Predicted
↪  Sweetness (RF)': best_model_prediction})
print(best_results_df.head())
plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
plt.scatter(best_results_df['Actual Sweetness'], best_results_df['Predicted
↪  Sweetness (RF)'], alpha=0.5, label='Best Predictions (RF)')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--',
↪  lw=2, label='Ideal Prediction')
plt.xlabel('Actual Sweetness')
plt.ylabel('Predicted Sweetness')
plt.title('Best SVR Model Predictions')
plt.legend()

# Practical Applications
# Agriculture:
    # Farmers can better determine the optimal harvest time for apples to
    ↪  ensure they meet desired sweetness levels,
    # improving market value and consumer acceptance.
```

```
    # Farmers can use our model to identify which traits most contribute to
    ↪ apple sweetness, guiding selective
    # breeding programs to produce sweeter apples naturally.
# Supply Chain Optimization:
    # The fruit produce industry can implement automatic sorting systems that
    ↪ use our model to classify apples by predicted
    # sweetness from external features. This gives insights into how different
    ↪ factors affect apple sweetness
    # can inform storage and distribution decisions, such as length and
    ↪ storage and transportation conditions and distance.
# Consumer Market:
    # Fruit companies can use sweetness predictions to design new apple-based
    ↪ products, providing target-specific customers
    # the desired taste and preferences for sweetness.
```

C Equations for SVR model

The Support Vector Regression (SVR) model equation with an RBF kernel is given by:

$$f(x) = \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) \cdot K(x_i, x) + b$$

where:

1. $f(x)$ is the predicted output for an input x.
2. N is the number of support vectors.
3. x_i are the support vectors.
4. α_i and α_i^* are the Lagrange multipliers associated with each support vector.
5. $K(x_i, x) = \exp(-\gamma \|x_i - x\|^2)$ is the RBF kernel function, which describes how input data is transformed into a higher-dimensional space to facilitate the linear separation of data points.
6. γ is the kernel coefficient, a parameter that needs to be set before the training process to control the shape of the kernel function.
7. b is the bias term, adjusting the output independently of the input features.

This equation represents how the SVR model predicts the output for a given input x, utilizing the support vectors, Lagrange multipliers, and the kernel function to capture complex, non-linear relationships within the data. The model's effectiveness and its generalization capability to unseen data are significantly influenced by the choice of the kernel function and the tuning of model parameters, including γ. The ϵ parameter, used during training to define the epsilon-insensitive loss, is implicit in the selection of support vectors and Lagrange multipliers but is not directly visible in the predictive function.

References

[1] U. S. D. of Agriculture, "Fresh apples, grapes, and pears: World markets and trade," Mar 2024.

[2] F. R. Harker, R. L. Amos, G. Echeverríaa, and F. A. Gunson, "Influence of texture on taste: Insights gained during studies of hardness, juiciness, and sweetness of apple fruit," *Journal of Food Science*, vol. 71, no. 2, pp. S77–S82, 2006.

[3] N. Elgiriyewithana, "Apple quality," Jan 2024.