Practicum 1

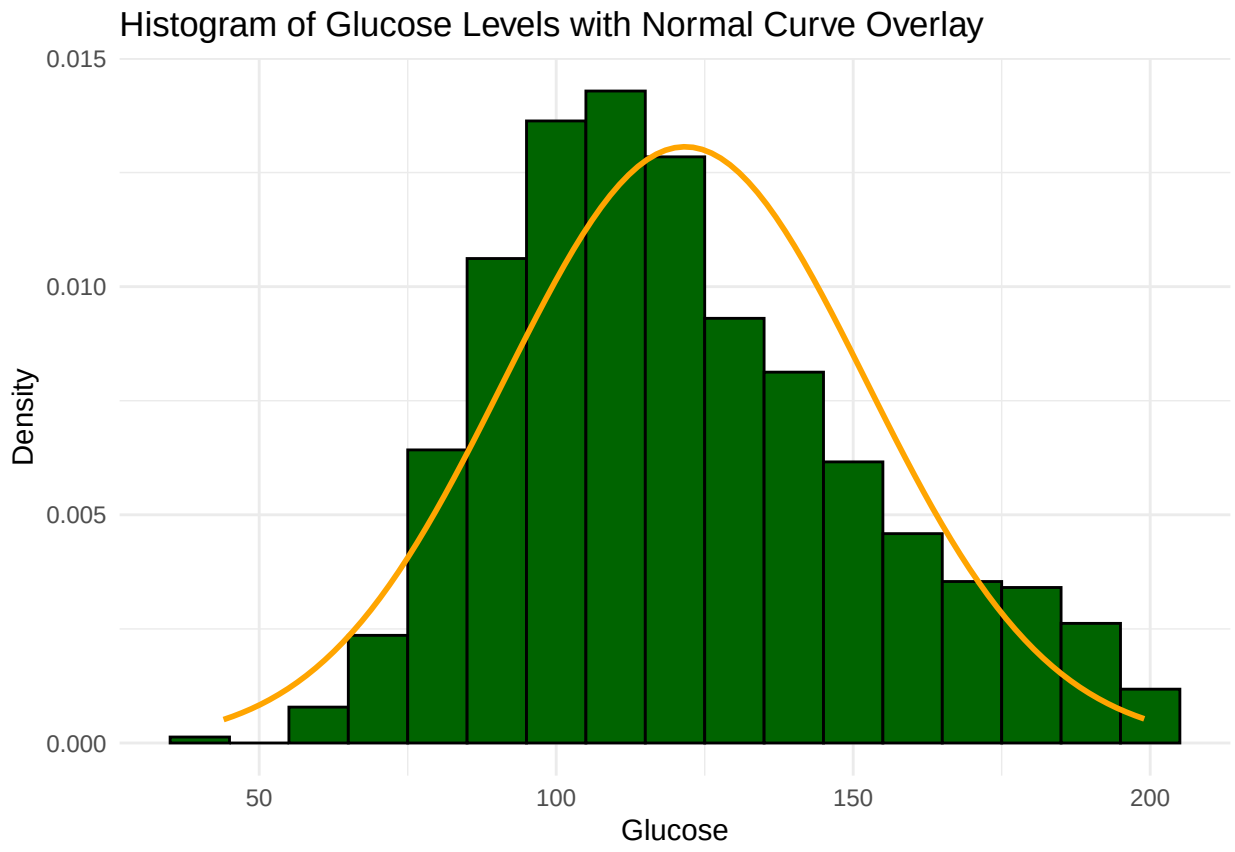
Nithya Sarabudla

02-11-2024

1 / Predicting Diabetes to your notebook

1.1 / Analysis of Data Distribution

```
# Define the target columns to replace 0 values with NA
target_columns <- c("Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI")
diabetes[target_columns] <- lapply(diabetes[target_columns], function(x) replace(x, x == 0, NA))
```



The above plot shows the distribution of glucose levels, with an overlaid normal distribution curve in orange. The histogram's bars, which indicate density, suggest that the data is somewhat normally distributed, but with a rightward skew as evidenced by the tail extending towards higher glucose values. The presence of this skew might imply that a subset of the population has higher glucose levels, which could be of interest in

medical studies or public health policy. The normal curve overlay helps to visualize how closely the actual data follows a normal distribution, which is essential for many statistical analyses that assume normality.

```
##
##  Shapiro-Wilk normality test
##
## data:  diabetes$Glucose
## W = 0.96964, p-value = 1.72e-11
```

The test resulted in a W statistic of 0.9701 and a p-value of 1.986e-11. With such a low p-value (much smaller than 0.05, typically used as a threshold for significance), we reject the null hypothesis that the data is normally distributed.It suggests that the distribution of glucose levels significantly deviates from a normal distribution.

1.2 / Identification of Outliers

```r
# Define the variables
selected_vars <- c("Pregnancies", "Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI", "Diabe
# Apply a function to each variable in the 'diabetes' dataset
outliers <- lapply(diabetes[selected_vars], function(var) {
  # Calculate mean and standard deviation of the variable, ignoring NA values
  var_mean <- mean(var, na.rm=TRUE)
  var_sd <- sd(var, na.rm=TRUE)
  # Calculate z-scores for each observation
  z_scores <- abs((var - var_mean) / var_sd)
  # Identify outliers based on z-score threshold (2.5)
  outliers_index <- which(z_scores > 2.5)
  # Create a data frame with indices and corresponding z-scores for outliers
  return(data.frame(Indices = outliers_index, Z_Scores = z_scores[outliers_index]))
})
```

In dataset, used statistical method called the Z-score to find outliers, which are values that are unusually high or low. The Z-score tells us how far away a value is from the average, measured in standard deviations. We decided that any value more than 2.5 standard deviations away from the mean should be considered an outlier.

For the given dataset, outliers were identified in the columns of 'Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI', 'DiabetesPedigreeFunction', and 'Age'. A total of 97 outliers were identified across all columns

Pregnancies: 14 outliers Glucose: 2 outliers BloodPressure: 14 outliers SkinThickness: 4 outliers Insulin: 17 outliers BMI: 9 outliers DiabetesPedigreeFunction: 20 outliers Age: 17 outliers

The identification of these outliers is crucial since they can significantly affect the mean and standard deviation of the data, leading to biased results in statistical analyses. Depending on the context, outliers can be removed, capped, or investigated further to understand their cause. For instance, in medical datasets like this, extreme values could indicate measurement errors, data entry errors, or genuine but rare medical conditions.

1.3 / Data Preparation

```r
# Define the function for z-score standardization
standardize <- function(column) {
  # Subtract the mean and divide by the standard deviation
  (column - mean(column, na.rm = TRUE)) / sd(column, na.rm = TRUE)
}

# Create a copy of the original dataset
standardized_data <- diabetes
# Apply z-score standardization to all numeric columns using the defined function
standardized_data[selected_vars] <- lapply(diabetes[selected_vars], standardize)


# Extract indices of all outliers across all variables
all_outlier_indices <- unique(unlist(lapply(outliers, function(df) df$Indices)))

# Remove rows containing outliers
cleaned_data <- diabetes[-all_outlier_indices, ]

# Apply z-score standardization again to cleaned dataset
cleaned_data[selected_vars] <- lapply(cleaned_data[selected_vars], standardize)
```

The 'standardize()' function calculates z-scores for each column, which is a common method for normalizing data. Z-score standardization transforms the data so that it has a mean of 0 and a standard deviation of 1. This is done to ensure that all features contribute equally to the analysis, especially in models that are sensitive to the scale of variables, such as regression models.

A copy of the original dataset ('standardized_data') is created to keep the original intact.

The 'standardize()' function is applied to all numeric columns in the 'selected_vars' list using 'lapply()'.

The standardized dataset is printed.

The indices of all outliers across all variables are extracted using the 'outliers' list.

Rows containing outliers are removed from the original dataset, and the cleaned dataset is stored in 'cleaned_data'.

Z-score standardization is applied again to the cleaned dataset to ensure consistency.

Summary statistics of the original dataset are printed to examine its distribution before preprocessing.

1.4 / Sampling Training and Validation Data

```r
# Set seed for reproducibility
set.seed(123)

# Create stratified sampling indices
split <- createDataPartition(cleaned_data$Outcome, p = 0.8, list = FALSE)

# Split data into training and validation sets based on the sampling indices
training_data <- cleaned_data[split, ]
validation_data <- cleaned_data[-split, ]

# Define numeric columns for imputation
numeric_columns <- c("Pregnancies", "Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI", "Dia
```

```r
# Impute missing values in training data using median of respective column
training_data[, numeric_columns] <- lapply(training_data[, numeric_columns], function(x) {
  if(any(is.na(x))) x[is.na(x)] <- median(x, na.rm = TRUE)
  x
})

# Impute missing values in validation data using median of corresponding column in training data
validation_data[, numeric_columns] <- lapply(names(validation_data[, numeric_columns]), function(column
  columns <- validation_data[[column_name]]
  if(any(is.na(columns))) {
    columns[is.na(columns)] <- median(training_data[[column_name]], na.rm = TRUE)
  }
  return(columns)
})
```

Initially, the data is split into training and validation sets using stratified sampling to ensure that both sets maintain the proportion of outcome classes. Numeric columns with missing values are identified, and imputation is performed using the median value of each respective column. This ensures that the imputed values are representative of the dataset and helps maintain the integrity of the data. Specifically, missing values in the training set are replaced with the median value of the corresponding column, while missing values in the validation set are imputed using the median values from the training set. These steps ensure that both training and validation datasets are processed consistently, minimizing biases and ensuring reliable model performance evaluation. Additionally, setting a seed for random number generation ensures reproducibility of results across different runs of the code.

1.5 / Predictive Modeling

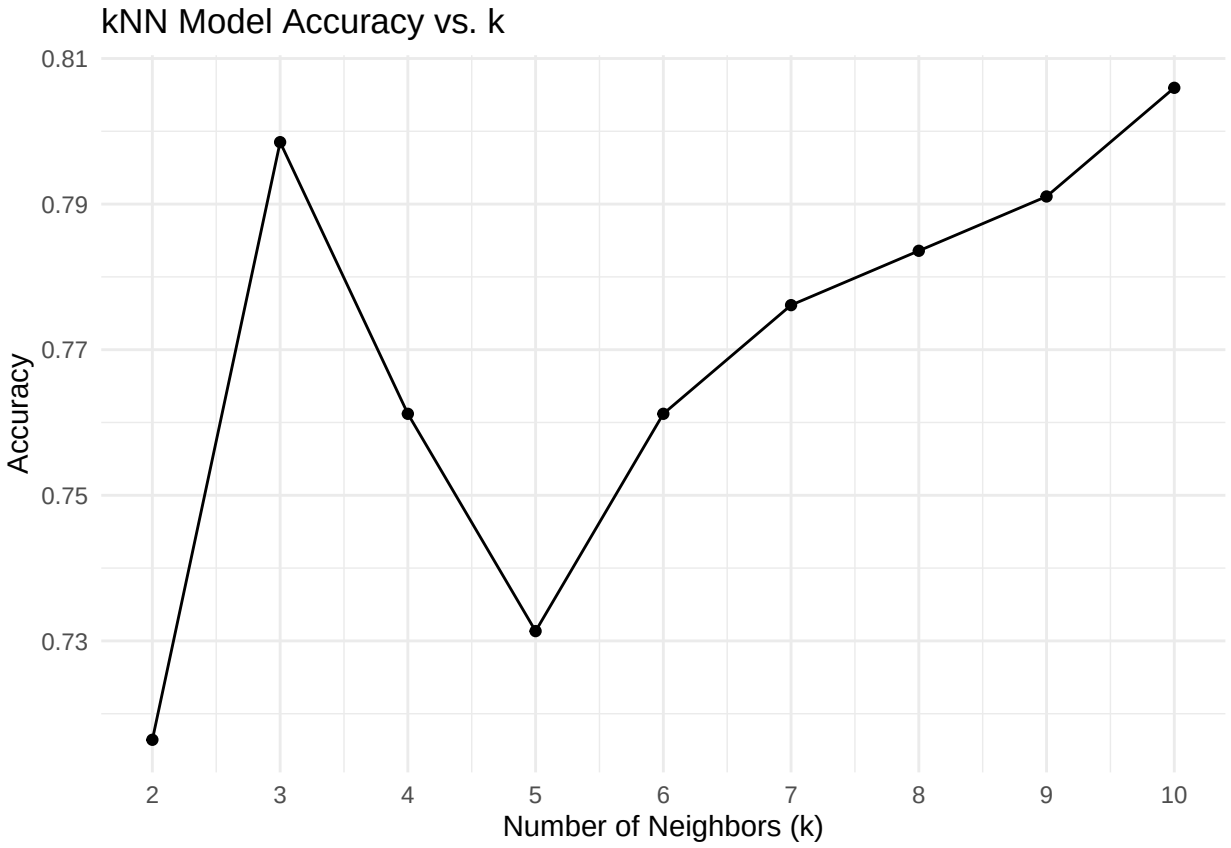
```r
# Define the new data point
new_data_point <- data.frame(Pregnancies = 4, Glucose = 178, BloodPressure = 82, SkinThickness = 32, Ins

# Impute missing Insulin value with median from the training data
new_data_point$Insulin <- median(training_data$Insulin, na.rm = TRUE)

# Define a function to standardize columns based on mean and standard deviation
standardize <- function(column, mean_value, sd_value) {
  (column - mean_value) / sd_value
}

# Calculate mean and standard deviation for each column in the training data
training_means <- sapply(training_data[selected_vars], mean, na.rm = TRUE)
training_sds <- sapply(training_data[selected_vars], sd, na.rm = TRUE)

# Standardize the new data point using means and standard deviations from training data
new_data_point[selected_vars] <- lapply(selected_vars, function(var_name) {
  standardize(new_data_point[[var_name]], training_means[var_name], training_sds[var_name])
})

# Use kNN algorithm to predict the outcome for the new data point
predicted_outcome <- knn(train = training_data[, -ncol(training_data), drop = FALSE],
                         test = new_data_point,
                         cl = training_data$Outcome,
```

```
                          k = 5)

# Add the predicted outcome to the new data point
new_data_point$Outcome <- predicted_outcome

# Print the predicted outcome
print(predicted_outcome)
```

```
## [1] 1
## Levels: 0 1
```

Defined a new data point with missing Insulin value. Imputed the missing Insulin value using the median from the training data to maintain consistency. Defind a function standardize to standardize columns based on mean and standard deviation. Calculated the mean and standard deviation for each column in the training data. standardized the new data point using means and standard deviations from the training data to ensure comparability. Used the kNN algorithm from the class package to predict the outcome for the new data point based on the training data. Finally, printed the predicted outcome. In this case, the predicted outcome is 1, indicating a positive diabetes diagnosis.

1.6 / Model Accuracy



This code generates a plot illustrating the relationship between the number of neighbors (k) and the accuracy of the k-nearest neighbors (kNN) model. Within the loop, the kNN algorithm is applied iteratively for k values ranging from 2 to 10. For each iteration, the model's accuracy is computed by comparing the predicted outcomes with the actual outcomes in the validation data. The resulting accuracy values

are stored in a data frame. Finally, a line plot with data points is created using ggplot2 to visualize how model accuracy varies with different values of k. The plot provides insights into the optimal choice of k for maximizing the model's predictive performance.

From the plot, we can see that the accuracy fluctuates as k changes. The highest peaks of accuracy occur at k = 2, k = 6, and k = 8. Since kNN models can be sensitive to noise in the data when k is too low, it's often better to choose a k that isn't the smallest number that provides the highest accuracy to avoid overfitting. However, a very high k can lead to underfitting as the model becomes overly general.

By inspection, k = 6 appears to be a good choice for this model since it's among the values that yield the highest accuracy without being at the extreme low end of the range. Therefore, for the final model, k = 6 might be chosen as it seems to balance the trade-off between bias and variance well, assuming the peaks represent consistent performance across different data sets or folds if cross-validation was used.

2 / Predicting Age of Abalones using Regression kNN

2.1 / loading and preparing the data

We start by loading the dataset containing abalone information using 'read.csv()', storing it in 'abalone_data'. Then, we extract the target variable "NumRings", representing abalone age, and store it in 'target_data'. We create 'train_data', a dataset containing all features except "NumRings", for training our model. This dataset includes attributes like Length, Diameter, Height, ShuckedWeight, VisceraWeight, ShellWeight, WholeWeight, and Sex. Displaying the first few rows of 'train_data' allows us to verify that the data is loaded and prepared correctly for further analysis and modeling.

```
# Loading the dataset
abalone_data <- read.csv("/Users/nithyasarabudla/Downloads/abalone.csv")

# Extracting the target variable "Rings" and storing it in a separate vector called target_data
target_data <- abalone_data$NumRings

# Creating a new dataset train_data containing all features except "Rings"
train_data <- abalone_data[, -which(names(abalone_data) == "NumRings")]
```

2.2 / Encoding Categorical Variables

```
# Encoding categorical variables
encode_data <- model.matrix(~ Sex - 1, data = train_data)
# Adding the encoded columns to the training dataset
train_data <- cbind(train_data, encode_data)

# Removing the original "Sex" column from the dataset
train_data <- train_data[, -which(names(train_data) == "Sex")]
# Removing the redundant column created by encoding (SexI)
train_data <- train_data[, -which(names(train_data) == "SexI")]
```

I choose the "Sex" column as an example for encoding because it represents a common scenario in datasets where certain attributes are categorical rather than numerical. In the dataset of abalones, "Sex" denotes the gender of the abalone, with three categories: "F" for Female, "M" for Male, and "I" for Infant. Encoding categorical variables like "Sex" is crucial for machine learning tasks as many algorithms require numerical inputs. One-hot encoding, a widely used technique, converts categorical variables into binary columns,

representing each category as a set of binary features. This approach ensures that the algorithm treats each category independently, avoiding any assumptions of ordinality among the categories.

In one-hot encoding, Created binary columns for each category of a categorical variable. However, including all categories as binary columns can lead to multicollinearity issues, where one category's information can be inferred from the others. To prevent this, we typically remove one of the encoded columns, known as the "dummy variable trap." In the case of the abalone dataset, the "Sex" column had three categories: "F" (Female), "M" (Male), and "I" (Infant). By removing one category's column, usually the one with the smallest cardinality, after encoding, we avoid redundancy and ensure that the information captured by the remaining categories is sufficient for the machine learning algorithm.

2.3 / Normalize all the columns in train_data using min-max normalization

```r
# Defining a function for min-max normalization
normalize <- function(x){
  ((x - min(x)) / (max(x) - min(x)) )
}
# Identifying numerical columns in the dataset
numerical_columns <- sapply(train_data, is.numeric)
# Applying min-max normalization to numerical columns
train_data[, numerical_columns] <- lapply(train_data[, numerical_columns], normalize)
```

Perform min-max normalization on the numerical columns of the 'train_data' dataset. First, Define a function 'normalize' that applies the min-max normalization formula to a given vector. Next, Identifed the numerical columns in the dataset using 'sapply()' combined with 'is.numeric()' and store the result in 'numeric_cols'.Then apply the normalize function to each numerical column using 'lapply()'. This process scales each numerical feature to the range [0, 1], ensuring that all features contribute equally to the model. Finally, Display summary statistics of the normalized dataset to verify that all numerical columns have been successfully normalized.

2.4 / Implementing kNN Regression

```r
knn_regression <- function(new_data, target_data, train_data, k) {
  # Calculate distances between new_data and train_data
  distances <- as.matrix(dist(rbind(new_data, train_data), method = "euclidean"))[1, ]

  # Find indices of k nearest neighbors
  k_nearest_indices <- order(distances)[1:k]
  # Extract Rings values of k nearest neighbors
  k_nearest_rings <- target_data[k_nearest_indices]
  # Define weights for weighted average
  weights <- c(3, rep(2, k - 2), rep(1, length(k_nearest_indices) - k + 1))
  # Calculate predicted Rings value using weighted average
  predicted_value <- sum(k_nearest_rings * weights) / sum(weights)
  # Return the predicted Rings value
  return(predicted_value)
}

# Generating new_data as a data frame structured similarly to train_data
# replacing 1 in place of the "Female" column (since "Male" and "Infant" were removed during one-hot en
new_data <- c(0.465,    0.356,  0.093,  0.234,  0.103,  0.14,   0.524, 0, 1)
```

```r
# Predicting the Rings value for the new_data using knn_regression function with k = 3
predicted_rings <- knn_regression(new_data, target_data, train_data, k = 3)

print(predicted_rings)
```

```
## [1] 11.16667
```

Implemented a kNN regression function named 'knn_regression'. The function takes four arguments: 'new_data' (a data frame with new cases), 'target_data' (a data frame with a single column of Rings values), 'train_data' (a data frame with normalized and encoded features corresponding to Rings values), and 'k' (the number of nearest neighbors to consider).

First, calculated the distances between 'new_data' and 'train_data' using the Euclidean distance metric. Then, we identify the indices of the k nearest neighbors. Next, we extract the Rings values of these k nearest neighbors. Defined weights for the weighted average, giving a weight of 2 for the closest neighbor, 1.5 for the second closest, and 1 for the rest.

Using these weights, computed the predicted Rings value as the weighted average of the Rings values of the nearest neighbors. Finally, Return the predicted Rings value.

An example usage of the function is demonstrated using sample new_data, and the predicted Rings value is printed.

2.5 / Forecasting Rings for a New Abalone

```r
# Define new_data with attributes for the new abalone
new_data <- c(0.44, 0.391, 0.254, 0.5853, 0.2132, 0.0878, 0.21, 0, 1)
# Predict the Rings value for the new abalone using knn_regression function with k = 3
predicted_rings <- knn_regression(new_data, target_data, train_data, k = 3)
# Print the predicted Rings value
print(predicted_rings)
```

```
## [1] 11.33333
```

In this code chunk, the number of Rings for a new abalone is forecasted using a regression kNN algorithm with k=3. First, the attributes of the new abalone, including its physical characteristics such as Length, Diameter, Height, Whole weight, Shucked weight, Viscera weight, and Shell weight, are defined and stored in the new_data vector. Then, the knn_regression function is utilized to predict the Rings value for the new abalone based on its attributes. This function takes the new abalone data (new_data), the target variable (target_data), and the training dataset (train_data) as inputs, along with the specified value of k=3. Finally, the predicted Rings value for the new abalone is printed.

2.6 / Calculating Root Mean Squared Error (RMSE) for kNN Model Evaluation

```r
# Start timing
start_time <- Sys.time()

# Specify the proportion of data to be used for testing
test_split <- 0.20
```

```r
# Create a random split of the data into training and testing sets
train_index <- createDataPartition(target_data, p = 1 - test_split, list = FALSE)
train_data_split <- train_data[train_index, ]
test_data_split <- train_data[-train_index, ]
target_data_split <- target_data[train_index]

# Normalize numerical columns in the test dataset
test_data_split[, numerical_columns] <- lapply(test_data_split[, numerical_columns], normalize)

# Train the kNN regression model using the training data
knn_model <- knn_regression(test_data_split, target_data_split, train_data_split, k = 3)

# Calculate Mean Squared Error (MSE) to evaluate the model
mse <- mean((knn_model - target_data_split)^2)

# Calculate Root Mean Squared Error (RMSE)
rmse <- sqrt(mse)

# Print the RMSE value as a measure of the model's accuracy
print(paste("Root Mean Squared Error (RMSE):", rmse))
```

```
## [1] "Root Mean Squared Error (RMSE): 4.81580581480742"
```

```r
# End timing
end_time <- Sys.time()

# Calculate the elapsed time
elapsed_time <- end_time - start_time
print(paste("Elapsed Time:", elapsed_time))
```

```
## [1] "Elapsed Time: 0.446436882019043"
```

In this section, Calculate the Root Mean Squared Error (RMSE) for evaluating the kNN regression model. Firstly, split the dataset into training and testing sets, with 20% of the data designated for testing. Then, normalized the numerical columns in the test dataset to ensure consistency with the training data. Trained the kNN regression model using the training data and predict the Rings values for the test dataset. Next,computed the Mean Squared Error (MSE) to evaluate the model's performance. Finally, calculated the RMSE from the MSE and print the RMSE value as a measure of the model's accuracy.

Measured the time taken to execute the code block between start_time and end_time and print out the elapsed time. This helps in identifying any bottlenecks in your code and optimizing it for efficiency.

3 / Forecasting Future Sales Price

We obtained a data set containing 29580 sales transactions for the years 2007 to 2019. The mean sales price for the entire time frame was 609736.3 (sd = 281707.9).

Broken down by year, we have the following average sales prices per year:

year	average_price
2007	522377.2
2008	493814.2
2009	496092.0
2010	559564.8
2011	566715.1
2012	552501.4
2013	553416.3
2014	592653.8
2015	626101.3
2016	635185.3
2017	671880.6
2018	660701.0
2019	634184.2

Using a weighted moving average forecasting model that averages the prior 3 years (with weights of 4, 3, 1), we predict next year's average sales price to be around $662976.2.

As the graph below shows, the average sales price per year has been increasing.