
# MARMARA UNIVERSITY
# FACULTY OF ENGINEERING

## CSE4074.1 - Machine Learning

### Term Project – Model Development Report

**Due Date:** 23.12.2024


| | Name Surname | Student Id |
|---|---|---|
| 1 | Ahmet Arda Nalbant | 150121004 |
| 2 | Umut Bayar | 150120043 |
| 3 | Niyazi Ozan Ateş | 150121991 |
| 4 | Murat Albayrak | 150120025 |
| 5 | Kadir Pekdemir | 150121069 |

# Model Development Description

## 1.

[124]:
```
## Begin of model development
## Since we will be working on different models, I copy the dataset into the objects I created with those model names.
df_LR=df.copy() ## Using for LinearRegression
df_RR=df.copy() ## Using for RidgeRegression
df_logistic=df.copy() ## Using for logistic Regression algorithm
df_cluster= df.copy() ## Using for cluster algorithm
```

•1.1 Purpose of Copying:

- Since different models (e.g., Linear Regression, Ridge Regression, Logistic Regression, Clustering) might **require** different preprocessing steps or modifications to the data, separate copies of the original dataset are created. This ensures that changes made for one model do not affect the datasets for other models.

• 1.2 Code Details:

- `df_LR = df.copy()`: Creates a copy of the original dataset and assigns it to `df_LR`, which will be used for **Linear Regression**.
- `df_RR = df.copy()`: Creates another copy for **Ridge Regression**.
- `df_logistic = df.copy()`: This copy will be used for **Logistic Regression**.
- `df_cluster = df.copy()`: Another copy is created for a **Clustering algorithm**.

• 1.3 Why `.copy()` is Used:

- Without `.copy()`, assigning `df_LR = df` would create a reference to the same object, meaning any changes to `df_LR` would directly affect `df`. Using `.copy()` ensures that these are independent copies of the dataset.

## 2.

```
## create linear regression model with using train test split

y = df_LR[["sellingprice"]]
x = df.drop(columns=["sellingprice"])
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=20)
lr= LinearRegression()
model_lr=lr.fit(x_train,y_train)
```

2.1 Define Target and Features:

- `y = df_LR[["sellingprice"]]`:
  - Extracts the target variable (`sellingprice`) from the dataset `df_LR` to predict.
  - It is kept as a DataFrame (with double brackets) to maintain a 2D structure.
- `x = df.drop(columns=["sellingprice"])`:
  - Removes the `sellingprice` column from the dataset `df` to use the remaining columns as features (`x`).

2.2 Split the Dataset:

- `train_test_split(x, y, test_size=0.2, random_state=20)`:
  - Splits the features (`x`) and target (`y`) into training and testing sets.
  - `test_size=0.2`: 20% of the data is reserved for testing, while 80% is used for training.
  - `random_state=20`: Sets a fixed seed for reproducibility, ensuring the same split occurs every time.
- The result:
  - `x_train`, `y_train`: Training data for model fitting.
  - `x_test`, `y_test`: Testing data for model evaluation.

2.3 Train the Linear Regression Model:

- `lr = LinearRegression()`:
  - Initializes a Linear Regression model instance.
- `model_lr = lr.fit(x_train, y_train)`:
  - Trains (or "fits") the Linear Regression model on the training data (`x_train` and `y_train`).
  - The trained model is stored in `model_lr`.

**3.**

```
## create ridge regression model with using train test split

x = df.drop(columns=["sellingprice"])
y = df["sellingprice"]

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=20)

ridge_model = Ridge(alpha=1.0)
ridge_model.fit(x_train, y_train)
```

3.1 Define Features and Target:

- `x = df.drop(columns=["sellingprice"])`:
  - Removes the `sellingprice` column from the dataset `df` to use the remaining columns as input features (`x`).
- `y = df["sellingprice"]`:
  - Selects the `sellingprice` column as the target variable (`y`) for prediction.

3.2 Split the Dataset:

- `train_test_split(x, y, test_size=0.2, random_state=20)`:
  - Splits the features (`x`) and target (`y`) into training and testing sets.
  - `test_size=0.2`: Reserves 20% of the data for testing and uses 80% for training.
  - `random_state=20`: Ensures reproducibility by fixing the random seed.
- Output:
  - `x_train`, `y_train`: Used to train the Ridge Regression model.
  - `x_test`, `y_test`: Used to test and evaluate the model's performance.

3.3 Create and Train the Ridge Regression Model:

- `ridge_model = Ridge(alpha=1.0)`:
  - Initializes a Ridge Regression model with an `alpha` value of 1.0.
  - Ridge Regression applies **L2 regularization**, which penalizes large coefficients to reduce overfitting and improve generalization.
  - The `alpha` parameter controls the strength of regularization:
    - Higher `alpha` values increase regularization, making the model more robust but potentially less flexible.
- `ridge_model.fit(x_train, y_train)`:
  - Fits the Ridge Regression model on the training data (`x_train` and `y_train`).

**4.**

```
## for using logistic models, our target column that is selling price categorized.
df_logistic['sellingprice_category'] = pd.cut(df_logistic['sellingprice'],
                                              bins=[0, 50000, 150000, 250000],
                                              labels=['Low Price', 'Medium Price', 'High Price'])
df_logistic.drop("sellingprice",axis=1)
```

4.1.Categorize the `sellingprice` Column:

- `pd.cut()`:
  - A pandas function used to divide continuous data into bins or intervals.
- Parameters:
  - `df_logistic['sellingprice']`: The column to be categorized.
  - `bins=[0, 50000, 150000, 250000]`: Specifies the boundaries of the bins.
    - Values between `0` and `50,000` are labeled as **Low Price**.
    - Values between `50,001` and `150,000` are labeled as **Medium Price**.
    - Values between `150,001` and `250,000` are labeled as **High Price**.
  - `labels=['Low Price', 'Medium Price', 'High Price']`: Assigns descriptive labels to the bins.

4.2 Drop the Original `sellingprice` Column:

- Removes the original `sellingprice` column from `df_logistic`, as the categorized column (`sellingprice_category`) will be used instead.
- `axis=1`: Specifies that the column (and not a row) is to be removed.

5.

```
## Logistic Regression
X = df_logistic.drop(columns=["sellingprice_category"])
y = df_logistic["sellingprice_category"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=20)

logistic_model = LogisticRegression(max_iter=1000)

logistic_model.fit(X_train, y_train)

predictions = logistic_model.predict(X_test)

accuracy = accuracy_score(y_test, predictions)
```

- 5.1 Define Features and Target:
  - `X = df_logistic.drop(columns=["sellingprice_category"])`: This line defines `X`, the input features, by removing the `sellingprice_category` column (which is the target variable for classification).
  - `y = df_logistic["sellingprice_category"]`: This assigns the `sellingprice_category` column as the target variable `y`.

- 5.2 Split the Dataset:
  - `train_test_split(X, y, test_size=0.2, random_state=20)`: The data is split into training and testing sets.
    - `X_train`, `y_train`: The training set used to train the model.
    - `X_test`, `y_test`: The testing set used to evaluate the model.
    - `test_size=0.2`: 20% of the data is reserved for testing, and 80% for training.
    - `random_state=20`: Ensures reproducibility of the train-test split.

- 5.3 Create and Train the Logistic Regression Model:
  - `logistic_model = LogisticRegression(max_iter=1000)`: Initializes a Logistic Regression model, setting `max_iter=1000` to ensure convergence during training, especially if the dataset is large or complex.
  - `logistic_model.fit(X_train, y_train)`: Trains the Logistic Regression model on the training data.

- 5.4 Make Predictions:
  - `predictions = logistic_model.predict(X_test)`: The model makes predictions on the testing data (`X_test`).

- 5.5 Evaluate the Model:
  - `accuracy = accuracy_score(y_test, predictions)`: The accuracy score of the model is calculated by comparing the predicted values (`predictions`) with the true values (`y_test`) from the testing set.

## 6.

```
## Random Forest Classifier model
df_RFC['sellingprice_category'] = pd.cut(df_RFC['sellingprice'],
                                         bins=[0, 50000, 150000, 250000],
                                         labels=['Low Price', 'Medium Price', 'High Price'])
df_RFC.drop("sellingprice",axis=1)

X = df_RFC.drop(columns=["sellingprice_category"])
y = df_RFC["sellingprice_category"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=20)

rf_model = RandomForestClassifier(n_estimators=100, random_state=20)

rf_model.fit(X_train, y_train)
```

- 6.1 Categorizing the Target Variable:
    - The `sellingprice` column is divided into three price categories: Low Price (0–50,000), Medium Price (50,001–150,000), and High Price (150,001–250,000). This is done to convert the continuous variable (`sellingprice`) into discrete categories, which is suitable for classification tasks.
- 6.2 Removing the Original `sellingprice` Column:
    - After categorizing the prices, the original `sellingprice` column is dropped from the dataset because it is no longer needed. The newly created `sellingprice_category` column will serve as the target variable for classification.
- 6.3 Defining Features and Target:
    - The features (`X`) are all the columns except the `sellingprice_category` (which is the target variable).
    - The target variable (`y`) is the `sellingprice_category`, which will be predicted by the Random Forest model.
- 6.4 Splitting the Dataset:
    - The data is divided into training and testing sets using a 80-20 split. The training set is used to train the model, while the testing set is used to evaluate the model's performance. The `random_state` ensures that the data split is reproducible.
- 6.5 Creating and Training the Random Forest Classifier:
    - A Random Forest model is initialized with 100 trees (estimators) and a fixed random seed to ensure consistent results.
    - The model is then trained on the training data (`X_train` and `y_train`), where it learns to classify the price categories based on the features.

## 7.

```
## Cluster Algorithms
## Apply k-means algorithm
df_cluster.head()
```

```
## define x and y values
X = df_cluster[['year', 'condition', 'odometer', 'mmr', 'sellingprice', 'make_encoded', 'model_encoded
        'trim_encoded', 'body_encoded', 'transmission_encoded', 'state_encoded', 'color_encoded',
        'interior_encoded', 'seller_encoded']]
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
inertia = []
for k in range(1, 11):  ## k values
    kmeans = KMeans(n_clusters=k, random_state=20)
    kmeans.fit(X_scaled)
    inertia.append(kmeans.inertia_)

# Elbow graphic
plt.plot(range(1, 11), inertia)
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()
```

```
## elbow point determined 4
kmeans = KMeans(n_clusters=4, random_state=20)
df_cluster['Cluster'] = kmeans.fit_predict(X_scaled)

# set result graphic
plt.figure(figsize=(8, 6))
plt.scatter(df_cluster['odometer'], df_cluster['sellingprice'], c=df_cluster['Cluster'], cmap='viridis
plt.title('K-means Clustering: Selling Price vs Odometer')
plt.xlabel('Odometer')
plt.ylabel('Selling Price')
plt.show()
cluster_summary = df_cluster.groupby('Cluster')[['year', 'condition', 'odometer', 'mmr', 'sellingprice
print(cluster_summary)
```

- 7.1 Feature Selection:
    - The code selects a set of features from the `df_cluster` dataframe, which includes both numerical and encoded categorical variables (e.g., `year`, `condition`, `odometer`, `sellingprice`, and various encoded features like `make_encoded`, `model_encoded`, etc.).
- 7.2 Feature Scaling:
    - **StandardScaler** is applied to standardize the feature set (`X`). This step is crucial because K-Means is sensitive to the scale of the data. Standardizing ensures that each feature contributes equally to the clustering process by transforming them to have a mean of 0 and a standard deviation of 1.

- 7.3 Finding the Optimal Number of Clusters Using the Elbow Method:

    - A loop iterates over values of `k` from 1 to 10. For each value of `k`, a K-Means model is trained, and the **inertia** (sum of squared distances from each point to its assigned cluster center) is recorded.
    - The inertia values are then plotted to form the **Elbow Curve**. The "elbow" point, where the inertia starts to level off, indicates the optimal number of clusters. In this case, the elbow point is determined to be at `k = 4`.

- 7.4 Applying K-Means with Optimal Clusters:

    - A K-Means model is initialized with 4 clusters (`n_clusters=4`), which is the optimal value identified from the elbow method.
    - The `fit_predict()` method is used to perform the clustering and assign each data point to a cluster. The resulting cluster assignments are added to the dataframe as a new column `Cluster`.

- 7.5 Visualization of Clusters:

    - A scatter plot is created to visualize the relationship between `odometer` (mileage) and `sellingprice`, with the color of each point representing its assigned cluster. This helps in visually interpreting how the data points are grouped based on these two features.

- 7.6 Cluster Summary:

    - The `groupby` method is used to calculate the mean values of the features (`year`, `condition`, `odometer`, `mmr`, `sellingprice`) for each of the 4 clusters. This summary provides insights into the characteristics of each cluster.