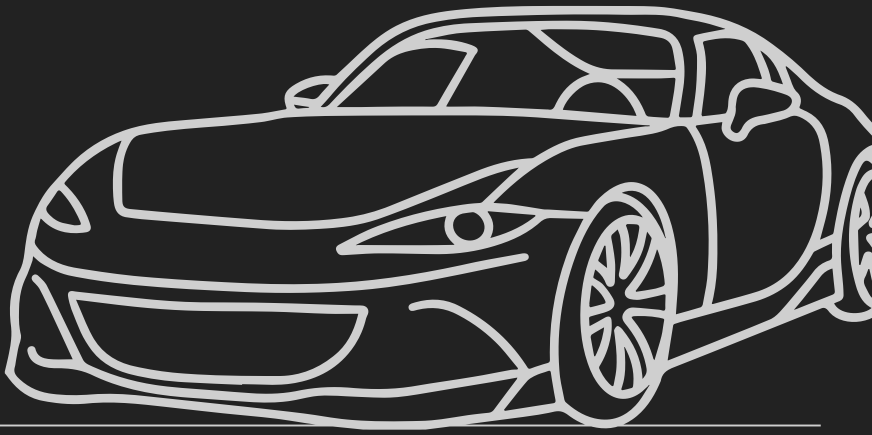
Vehicle Data Analysis

Matt Feinstein
Chris Gliatto



Step 1: Motive

Questions we want to answer about the vehicle market:

1. Is there correlation between the features of a vehicle (i.e. make, model, number produced) and its rate of getting stolen? If so, what features are most important?

1. How do the features of a specific vehicle (i.e. mileage, year produced, fuel type, etc) affect its selling price?

Step 2: Data Collection

<u>Individual Car Data</u>	A data set of individual vehicles and their features: mileage, fuel type, transmission, etc. The label is the selling price of the car.
<u>Vehicle Theft Data</u>	A data set of different car makes, related features (year, model, type, number produced). The label is the rate stolen. <u>(No available CSV)</u>

Step 2a: Web Scraping

```python
import requests
from bs4 import BeautifulSoup
import pandas as pd

base_url = 'https://www.nhtsa.gov/vehicle-theft-data?field_manufacturer_target_id=All&field_theft_type_value=All&field_theftyear_value_1=All&order=field_theft_rate&sort=desc'

data = []
headers = ['year', 'manufacturer', 'make', 'make/model', 'thefts', 'production', 'rate', 'type']

for page in range(0, 121):
    print(f"Scraping page {page}...")
    response = requests.get(f"{base_url}&page={page}")
    soup = BeautifulSoup(response.text, 'html.parser')

    table = soup.find('table', {'class': 'cols-8 table d8-port views-table'})
    if table:
        rows = table.find_all('tr')
        for row in rows:
            cells = [cell.text.strip() for cell in row.find_all('td')]
            if cells:
                data.append(cells)
    else:
        print('No table found')

if data:
    df = pd.DataFrame(data, columns=headers)
    # df.to_csv('vehicle_theft_data.csv', index=False)
    print("Data has been saved to 'vehicle_theft_data.csv'.")
```

Step 3: Data Analysis Part A

Naive Bayes+Information Gain(Car Theft Analysis)

Naive Bayes was used for the car theft analysis because it handles categorical data effectively, such as manufacturer, make, type, and production rate. Additionally, we incorporated information gain, which helped determine the most relevant features for predicting car theft risk. By leveraging the probabilistic relationships between these features and theft likelihood, it flags vehicles at higher risk based on their characteristics.

Naive Bayes Implementation

```python
df = pd.read_csv('data/vehicle_theft_data.csv', header=0)
features = ['year', 'manufacturer', 'make', 'make/model', 'production', 'rate', 'type']
df = df[features]

ave_rate = df['rate'].mean()
prod_deciles = df.production.quantile([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1])

test_df = df.sample(frac=0.25)      ChrisGiliotto, 2 hours ago • Commit #6
train_df = df

train_partition = create_partition(train_df, ave_rate, prod_deciles)
test_partition = create_partition(test_df, ave_rate, prod_deciles)

model = NaiveBayes(train_partition)
```

```python
def create_partition(df, ave_rate, prod_deciles):
```

```python
if f == 'production':
    for i in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]:
        if row[f] <= prod_deciles[i]:
            features[f] = i
            break
```

```python
label = 1 if row['rate'] >= ave_rate else 0
```

```python
class NaiveBayes:

    def __init__(self, part):

        self.K = part.K
        self.n = part.n
        self.F = part.F

        self.K_count = [0 for i in range(self.K)]
        self.f_count = [OrderedDict() for i in range(self.K)]

        # Construct feature counts with 0s
        for dict in self.f_count:
            for feature in list(self.F):
                dict[feature] = OrderedDict()
                for value in list(self.F[feature]):
                    dict[feature][value] = 0

        # Fill K_count and f_count variables
        for example in part.data:
            self.K_count[example.label] += 1
            for feature in example.features:
                self.f_count[example.label][feature][example.features[feature]] += 1

    def classify(self, x_test):

        probs = []
        for k in range(self.K):
            # Theta_k
            sum = log(self.K_count[k] + 1) - log(self.n + self.K)
            for feature in list(x_test):
                # Prod of Theta_k_i_v
                sum += log(self.f_count[k][feature][x_test[feature]] + 1) - log(self.K_count[k] + len(self.F[feature]))
            probs.append(sum)

        return probs.index(max(probs))
```

Information Gain Implementation

```python
def best_feature(self):

    # Class probabilities
    num_pos = 0
    for example in self.data:
        if example.label == 1:
            num_pos += 1

    class_prob = [(self.n-num_pos)/self.n, num_pos/self.n]


    # Entropy Calculation
    H = sum([-prob*log2(prob) for prob in class_prob])

    # Conditional Entropy Calculations (Using helper)
    con_H = {}
    for feature in self.F:
        con_H[feature] = self.feature_entropy(feature)

    # Convert to Gain
    gain = {}
    for feature in self.F:
        gain[feature] = H - con_H[feature]

    # Info printout
    print('\nInfo Gain:')
    for feature in gain:
        print(f'{feature}, {round(gain[feature], 6)}')
    print()

    # Return best feature from max gain
    best_f = max(gain, key=gain.get)
    return best_f
```

```python
# H(Y | X)
def feature_entropy(self, feature):

    sum = 0
    for val in self.F[feature]:
        count = 0
        for example in self.data:
            if example.features[feature] == val:
                count += 1
        sum += count/self.n * self.value_entropy(feature, val)

    return sum

# H(Y | X = v)
def value_entropy(self, feature, val):

    sum = 0
    for k in [-1, 1]:
        num = 0
        denom = 0
        for example in self.data:
            if example.features[feature] == val:
                denom += 1
                if example.label == k:
                    num += 1
        prob = 0
        if denom > 0:
            prob = num/denom
        if prob > 0:
            sum -= prob * log2(prob)

    return sum
```

Results

	0	1
0	319	62
1	44	178

Accuracy: 82.421% (497/603)

```
Info Gain:
year, 0.449583
manufacturer, 0.489055
make, 0.500906
make/model, 0.709026
production, 0.443928
type, 0.424551

Best feature: make/model
```

Step 3: Data Analysis Part B

Linear Regression(Car Pricing Analysis)

Linear regression was chosen for the car purchasing analysis because the dataset is numeric and aims to predict a continuous outcome: the selling price. With features like the car's year, price, and kilometers driven, linear regression models the relationship between these variables and the price, helping predict future values and understand factors like depreciation or transmission type. This method is ideal for uncovering pricing trends and aiding decision-making.

Linear Regression Implementation

```
> PriceRegression > Cardataanalysis.py > ...
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 1. Load and Preprocess the Data
def load_and_preprocess_data(file_path):
    df = pd.read_csv(file_path)

    # Encoding categorical columns
    df['Fuel_Type'] = df['Fuel_Type'].map({'Petrol': 0, 'Diesel': 1, 'CNG': 2})
    df['Seller_Type'] = df['Seller_Type'].map({'Dealer': 0, 'Individual': 1})
    df['Transmission'] = df['Transmission'].map({'Manual': 0, 'Automatic': 1})

    # Fill missing values with median for numeric columns only
    numeric_cols = df.select_dtypes(include=[np.number]).columns
    df[numeric_cols] = df[numeric_cols].fillna(df[numeric_cols].median())

    # Select features and target variable
    X = df[['Year', 'Present_Price', 'Kms_Driven', 'Fuel_Type', 'Seller_Type', 'Transmission']]
    y = df['Selling_Price']

    return X, y
# 2. Feature Scaling (Standardization)
def scale_features(X):
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)
    X_scaled = np.c_[np.ones((X_scaled.shape[0], 1)), X_scaled]  # Add bias term
    return X_scaled
```

```
# 3. Gradient Descent for Linear Regression
def gradient_descent(X_train, y_train, alpha=0.01, iterations=1000):
    m = len(y_train)
    theta = np.zeros(X_train.shape[1])

    cost_history = []

    for _ in range(iterations):
        # Hypothesis
        y_pred = np.dot(X_train, theta)

        # Cost function
        cost = (1 / (2 * m)) * np.sum((y_pred - y_train) ** 2)
        cost_history.append(cost)

        # Gradients
        gradients = (1 / m) * np.dot(X_train.T, (y_pred - y_train))

        # Update parameters
        theta -= alpha * gradients

    return theta, cost_history
```

Results

Mean Squared Error on Test Set: 3.4140111247113323

Learned Parameters(theta)

- Intercept (4.71): Base selling price when all features are zero.
- Year (1.09): Newer cars have higher selling prices.
- Present_Price (3.75): The current price of the car is strongly related to its selling price.
- Kms_Driven (-0.22): More kilometers driven results in a lower selling price.
- Fuel_Type (0.59): Diesel cars tend to be more expensive than Petrol cars.
- Seller_Type (-0.61): Cars sold by individuals tend to be cheaper than those sold by dealers.
- Transmission (0.56): Automatic cars tend to have a higher selling price than Manual cars

Results pt 2

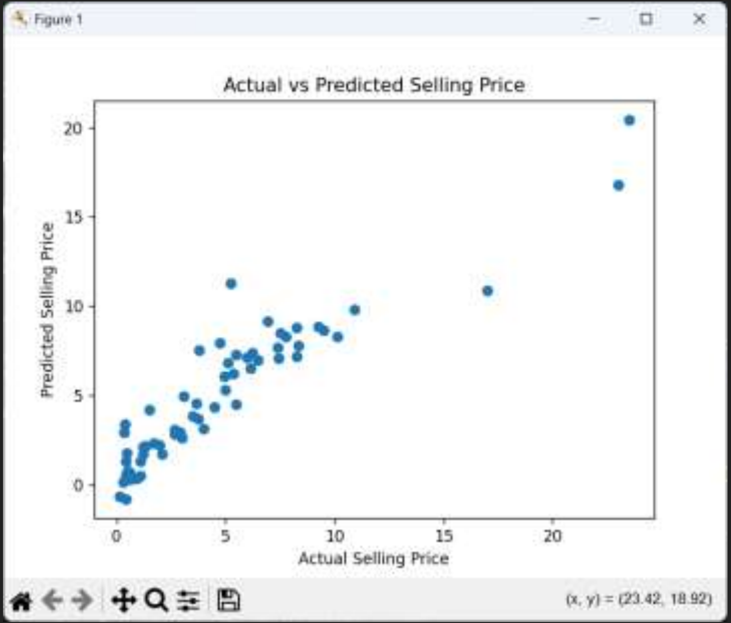


Figure 1
Actual vs Predicted Selling Price
Predicted Selling Price
Actual Selling Price
0
5
10
15
20
(x, y) = (23.42, 18.92)

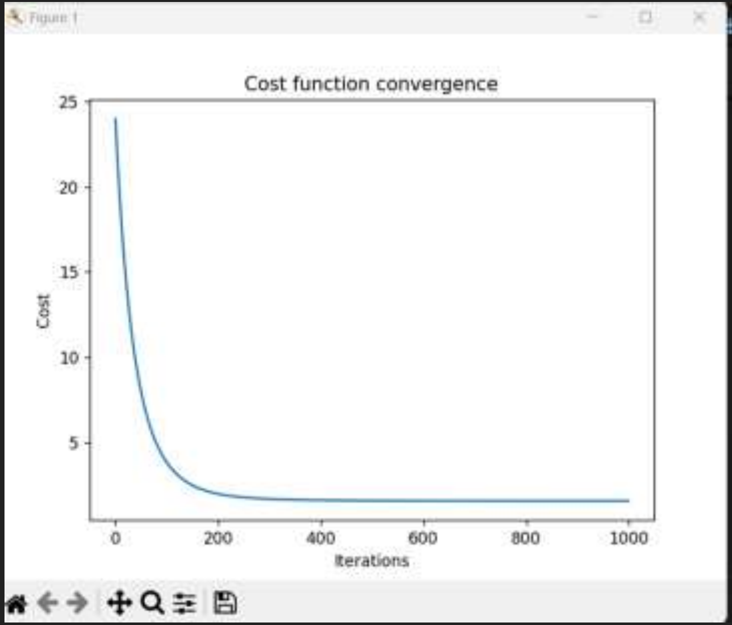


Figure 1
Cost function convergence
Cost
Iterations
5
10
15
20
25
0
200
400
600
800
1000

Works Cited

Bale, Rahul. Cars Selling Price. Kaggle, 2021, www.kaggle.com/code/rahulbale/cars-selling-price/input. Accessed 15 Nov. 2024.

National Highway Traffic Safety Administration. Vehicle Theft Data. U.S. Department of Transportation, https://www.nhtsa.gov/road-safety/vehicle-theft-prevention/theft-rates. Accessed 15 Nov. 2024.