Regression

GitHub Link: https://github.com/ZohoorAlmalki/ML_HWs.git

Introduction: [3]

The insurer needs access to previous data to estimate each user's medical expenses in order to create the best medical insurance packages. A medical insurer can use this information to plan a specific insurance outcome, manage large portfolios, or create more accurate pricing models. Accurately estimating insurance costs is the goal in each of these situations.

Dataset: [3,2]

1,339 medical insurance records are included in this dataset. The **target** variable **charges** are the individual medical expenses that are invoiced by health insurance; the remaining columns include personal data, like age, gender, family status, and if the patient smokes, among other characteristics.

Columns

- `age` : age of primary beneficiary
- `sex` : insurance contractor gender, female, male
- `bmi` : Body mass index, providing an understanding of body, weights that are relatively high or low relative to height,

objective index of body weight (kg / m ^ 2) using the ratio of height to weight, ideally 18.5 to 24.9

- `children` : Number of children covered by health insurance / Number of dependents
- `smoker` : Smoking
- `region` : the beneficiary's residential area in the US, northeast, southeast, southwest, northwest.
- `charges` : Individual medical costs billed by health insurance => This is my target.

In [5]:
```
import pandas as pd
import numpy as np
```

In [6]:
```
insurance=pd.read_csv('./Dataset/insurance.csv')
```

Next will show the first 5 samples in the dataset

In [8]:
```
insurance.head()
```

Out[8]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

Show the last 5 samples in the dataset

In [10]:
```
insurance.tail()
```

Out[10]:

	age	sex	bmi	children	smoker	region	charges
1333	50	male	30.97	3	no	northwest	10600.5483
1334	18	female	31.92	0	no	northeast	2205.9808
1335	18	female	36.85	0	no	southeast	1629.8335
1336	21	female	25.80	0	no	southwest	2007.9450
1337	61	female	29.07	0	yes	northwest	29141.3603

---

the dataset have a `1338 sample` each sample have a `7 features`.

In [12]:
```
insurance.shape
```

Out[12]:
```
(1338, 7)
```

We have to check if our dataset have a `null` values.

In [14]:
```
insurance.isna().sum()
```

Out[14]:
```
age         0
sex         0
bmi         0
children    0
smoker      0
region      0
charges     0
dtype: int64
```

there is no `null` values so we will not drop a column

Now we will describe the dataset

In [17]:
```
insurance.describe()
```

Out[17]:

	age	bmi	children	charges
count	1338.000000	1338.000000	1338.000000	1338.000000
mean	39.207025	30.663397	1.094918	13270.422265
std	14.049960	6.098187	1.205493	12110.011237
min	18.000000	15.960000	0.000000	1121.873900
25%	27.000000	26.296250	0.000000	4740.287150
50%	39.000000	30.400000	1.000000	9382.033000
75%	51.000000	34.693750	2.000000	16639.912515
max	64.000000	53.130000	5.000000	63770.428010

We will observe here the data type for each feature; and each non numeric values we have to encode it into a numeric form to be understandable for our model.

In [19]:
```
insurance.dtypes
```

Out[19]:
```
age           int64
sex          object
bmi         float64
children      int64
smoker       object
region       object
charges     float64
dtype: object
```

In [20]:
```
insurance['region'].value_counts()
```

Out[20]:
```
region
southeast    364
southwest    325
northwest    325
northeast    324
Name: count, dtype: int64
```

`pd.get_dummies()` is a Pandas function used to convert categorical variables into dummy/indicator variables (also known as one-hot encoding).

It creates new binary columns for each category (e.g., `sex_male`, `smoker_yes`, `region_northwest`, etc.).

`columns=catColumns` tells it to apply this transformation only to the `sex`, `smoker`, and `region` columns.

`drop_first=True` avoids the dummy variable trap (multicollinearity in regression) by dropping the first category of each column. For example:

Instead of creating both `sex_female` and `sex_male`, it might only create `sex_male`. If `sex_male` is 0, it means the person is female.

Similarly, it drops the first value for each categorical column.

region has four categories: `southeast`, `southwest`, `northwest`, and `northeast`. So `get_dummies()` creates one column for each category

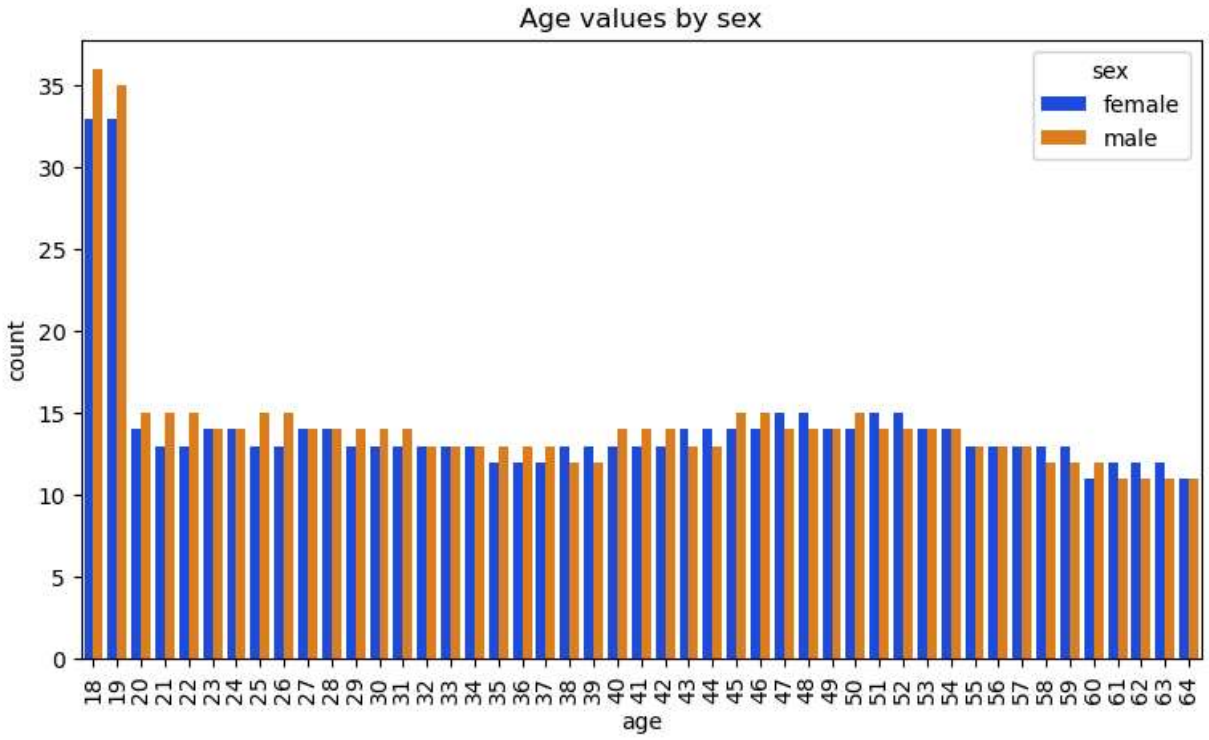
In [22]:
```
catColumns = ['sex', 'smoker', 'region']
insurance_dum = pd.get_dummies(insurance, columns = catColumns, drop_first=True)
insurance_dum.head()
```

Out[22]:

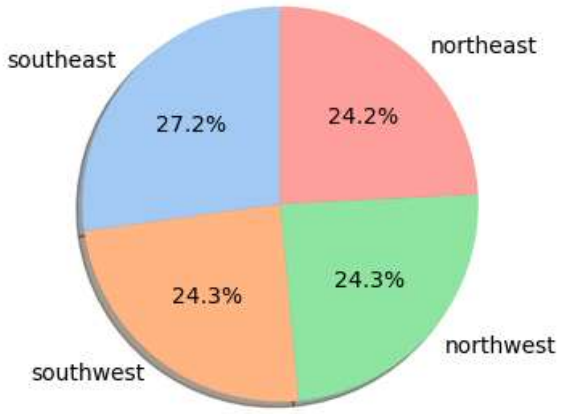
	age	bmi	children	charges	sex_male	smoker_yes	region_northwest	region_southeast	region_southwest
0	19	27.900	0	16884.92400	False	True	False	False	True
1	18	33.770	1	1725.55230	True	False	False	True	False
2	28	33.000	3	4449.46200	True	False	False	True	False
3	33	22.705	0	21984.47061	True	False	True	False	False
4	32	28.880	0	3866.85520	True	False	True	False	False

Data visualization [1]

In [25]:
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_palette('bright')
plt.figure(figsize=(9,5))
sns.countplot(x='age',hue='sex' ,data=insurance)
plt.title('Age values by sex')
plt.xticks(rotation=90)
plt.show()
```

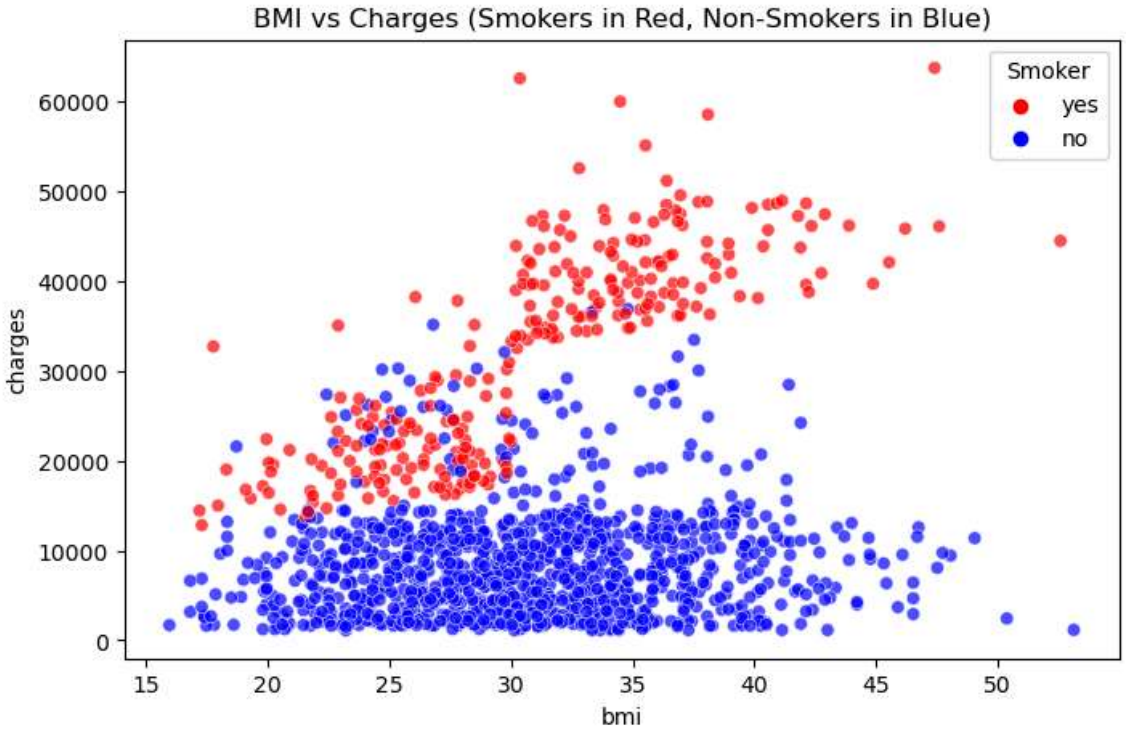


In [26]:
```
region=insurance['region'].value_counts()

labels=region.index
sizes=region.values
plt.figure(figsize=(6,4))
colors=sns.color_palette('pastel')
plt.pie(sizes,labels=labels,autopct='%1.1f%%',
        shadow=True,colors=colors,startangle=90)
plt.show()
```



In [28]:
```
plt.figure(figsize=(6,4))
labels=['Non-Smoker','Smoker']
size=insurance['smoker'].value_counts()
colors=['#99ff99','#ffcc99']
explode=(0,0.1)
plt.pie(size,labels=labels,colors=colors,explode=explode,autopct='%2.f%%')
circle = plt.Circle( (0,0),0.6, color='white')
p=plt.gcf()
p.gca().add_artist(circle)
plt.show()
```

In [30]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Set the figure size
plt.figure(figsize=(8, 5))

# Create a scatter plot with color based on 'smoker'
sns.scatterplot(
    x='bmi',
    y='charges',
    hue='smoker',  # Color points by smoker status
    palette={'yes': 'red', 'no': 'blue'},  # Map 'yes' to red, 'no' to blue
    data=insurance,
    alpha=0.7  # Slightly transparent for better visibility
)

# Add title and adjust legend
plt.title('BMI vs Charges (Smokers in Red, Non-Smokers in Blue)')
plt.legend(title='Smoker')  # Show legend with title

plt.show()
```

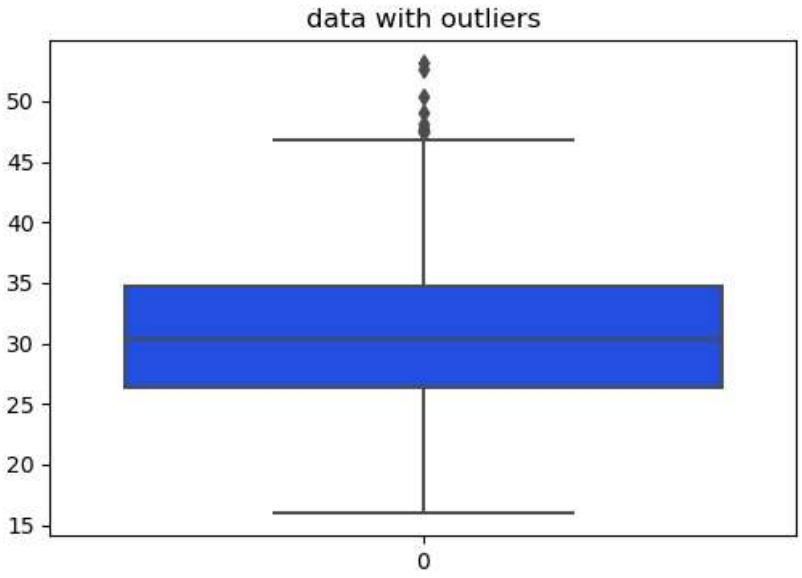


---

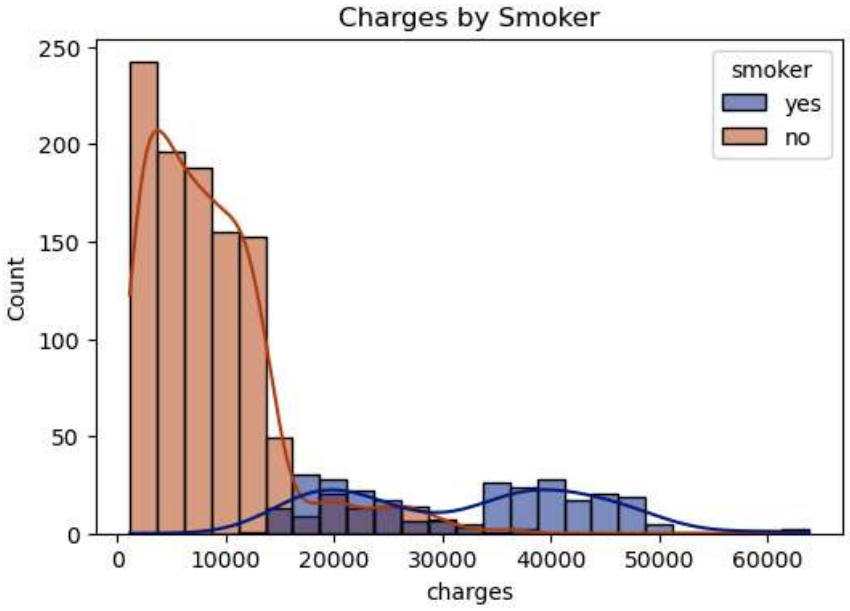
find outliers

In [32]:
```
plt.figure(figsize=(6,4))
sns.boxplot(insurance['bmi'])
```

```
plt.title('data with outliers')
plt.show()
```



In [34]:
```
# this part is corrected by chatgpt on this link below
# https://chatgpt.com/share/67fd8726-c634-8009-b3ee-1fea298abfb4
import warnings
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Clean the data anyway (always good practice)
insurance_clean = insurance.replace([np.inf, -np.inf], np.nan).dropna(subset=['charges'])

# Suppress specific FutureWarning
with warnings.catch_warnings():
    warnings.simplefilter("ignore", FutureWarning)

    fig, ax1 = plt.subplots(figsize=(6, 4))
    sns.set_palette('dark')
    sns.histplot(data=insurance_clean, x='charges', ax=ax1, bins=25, hue='smoker', kde=True)
    plt.title('Charges by Smoker')
    plt.show()
```



Methodology

Requirements:

You should write your code to investigate the following:

- A The impact of altering the cost function in linear regression.
  As you should write your own linear regression model and gradient descent code, you will compare the different reported cost functions:
  - $J(\theta) = \frac{1}{2} \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right)^2$
  - $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right)^2$
- The impact of learning rate on the training process

Try the learning rates (α) listed below, plot the value of $J(\theta)$ after each iteration and compare these plots and comment on them. - α = {0.0001, 0.001, 0.01, 0.1, 1, 10, 100}

- Your linear regression model vs sklearn

Compare your best model's performance with sklearn

- Regularized linear regression

Implement one of the regularization techniques on your linear regression model

- Compare your results with any other regressor of your choice (You can use a

library regressor in this part)

- Use a polynomial regression of any degree/transformation you choose.
- Discuss the performance with your model and sklearn in C).

Linear regression model and gradient descent code

In [38]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Hypothesis function
def predict(X, theta):
    return X @ theta

def cost_function_1(X, y, theta):
    m = len(y)
    error = predict(X, theta) - y
    return (0.5) * np.sum(error ** 2)

# Cost Function 2: With 1/m
def cost_function_2(X, y, theta):
    m = len(y)
    error = predict(X, theta) - y
    return (1 / (2 * m)) * np.sum(error ** 2)


# Gradient Descent
def gradient_descent(X, y, theta, alpha, iterations, cost_fn):
    cost_history = []
    m = len(y)

    for i in range(iterations):
        error = predict(X, theta) - y
        gradient = X.T @ error # Vectorized Gradient
        if cost_fn == 1:
            gradient = gradient / m # i know the derivative for cost fun1 is not divided by m like the cost fun2; but i t
            theta -= alpha * gradient
            #C:\Users\zohoo\AppData\Roaming\Python\Python311\site-packages\numpy\core\fromnumeric.py:88: RuntimeWarning:
            #   return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
            # C:\Users\zohoo\AppData\Local\Temp\ipykernel_11732\4075407248.py:13: RuntimeWarning: overflow encountered in
            #   return (0.5) * np.sum(error ** 2)
            # Stopped early at iteration 65, alpha=0.01 due to overflow.
            # Stopped early at iteration 1, alpha=0.01 due to divergence.
            # Stopped early at iteration 76, alpha=0.1 due to overflow.
            # Stopped early at iteration 0, alpha=0.1 due to divergence.
            # Stopped early at iteration 50, alpha=1 due to overflow.
            # Stopped early at iteration 0, alpha=1 due to divergence.
            # and the only way to avoid it to divide by m or not applay learning rate 0.01, 0.1 ,1
```

```
        else:
            gradient = gradient / m
            theta -= alpha * gradient
        # Compute cost using selected cost function
        cost = cost_function_1(X, y, theta) if cost_fn == 1 else cost_function_2(X, y, theta)

        # Stop if cost diverges
        if np.isnan(cost) or np.isinf(cost):
            print(f"Stopped early at iteration {i}, alpha={alpha} due to overflow.")
            break

        cost_history.append(cost)

    return theta, cost_history

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np

# 1. Extract features and labels (no bias yet)
X = insurance_dum[['age', 'bmi', 'smoker_yes']].values.astype(float)
y = insurance_dum['charges'].values.reshape(-1, 1).astype(float)

# 2. Scale features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 3. Split: first into train (60%) and temp (40%)
X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.4, random_state=42)

# Then split temp into validation (20%) and test (20%)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# 4. Add bias column to each set
def add_bias(X):
    return np.hstack([np.ones((X.shape[0], 1)), X])

X_train_b = add_bias(X_train)
X_val_b = add_bias(X_val)
X_test_b = add_bias(X_test)

# Optional: print shapes
print("Train:", X_train_b.shape, y_train.shape)
print("Validation:", X_val_b.shape, y_val.shape)
print("Test:", X_test_b.shape, y_test.shape)

# Learning rates to try
alphas = [0.0001, 0.001, 0.01, 0.1, 1] # Learning rates with 10, 100 doesn't work for me (overflow encountered erorr)
#; so i keep those learning and discard the others
# so i just keep [0.0001, 0.001, 0.01, 0.1, 1] in my experiments
# also i know i am not achieving the HW requirements ; but i do my best :)
iterations = 100
```

```
Train: (802, 4) (802, 1)
Validation: (268, 4) (268, 1)
Test: (268, 4) (268, 1)
```

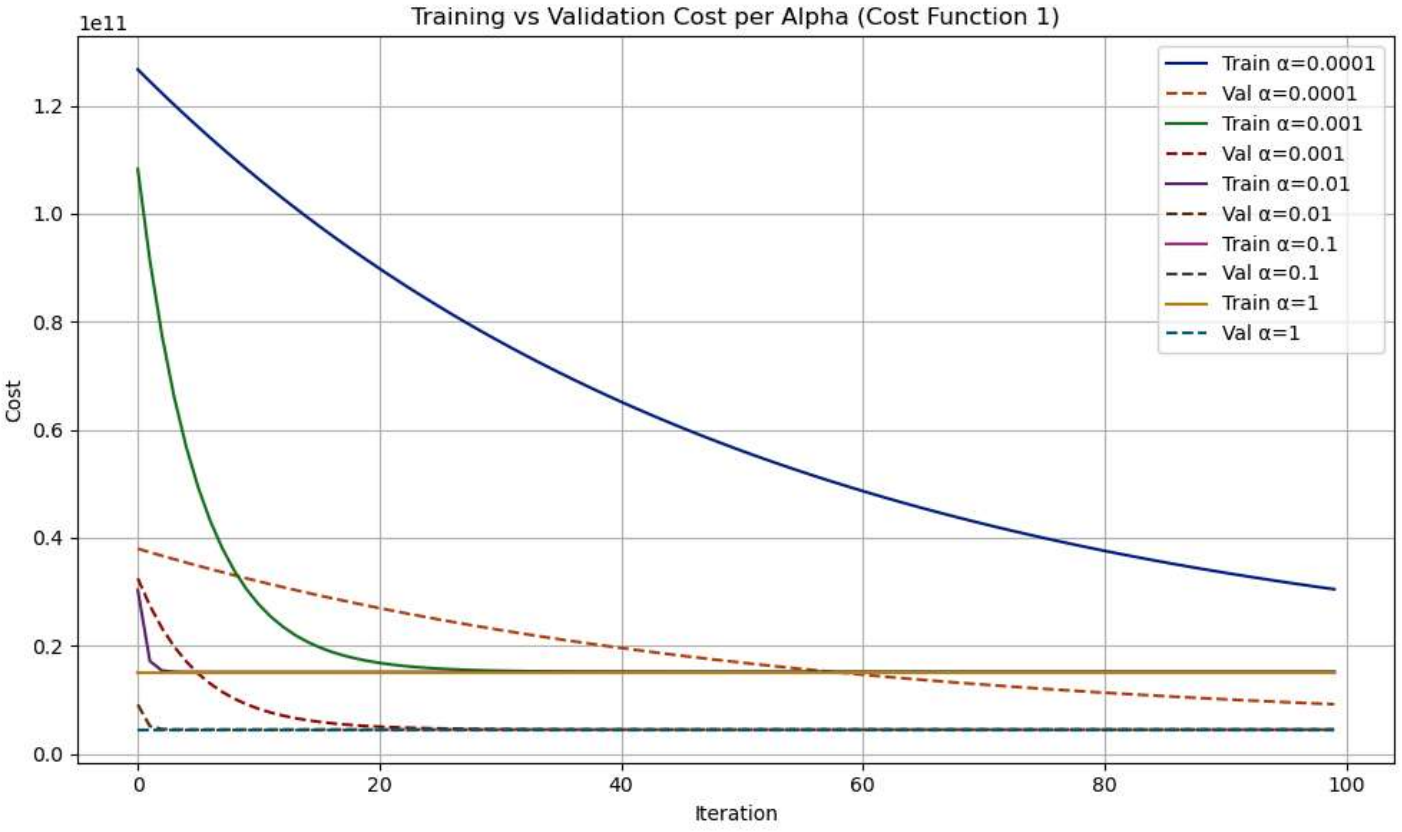
Model Training and validation:

In [40]:

```
# chatgpt generation codes  i just adaptive to my privious code
cost_fn = 1
best_val_cost1 = float('inf')
best_alpha1 = None
best_theta1 = None

train_histories = {}
val_histories = {}

for alpha in alphas:
    theta_init = np.zeros((X_train_b.shape[1], 1))
    theta = theta_init.copy()
    train_costs = []
    val_costs = []

    # Manual training loop to collect validation costs per iteration
    for i in range(iterations):

        theta, cost_history = gradient_descent(X_train_b, y_train, theta, alpha, iterations,cost_fn=1) # i add this to ca
```

```
        # first code do the learning here

        train_cost = cost_function_1(X_train_b, y_train, theta)
        val_cost = cost_function_1(X_val_b, y_val, theta)

        train_costs.append(train_cost)
        val_costs.append(val_cost)

    # Store histories
    train_histories[alpha] = train_costs
    val_histories[alpha] = val_costs

    # Track best model by lowest final val cost
    if len(val_costs) > 0 and val_costs[-1] < best_val_cost1:
        best_val_cost1 = val_costs[-1]
        best_alpha1 = alpha
        best_theta1 = theta


print(f'best_val_cost1 = {best_val_cost1}')
print(f'best_alpha1 = { best_alpha1}')
print(f'best_theta1 = {best_theta1}')


# Plot training vs validation costs
plt.figure(figsize=(10, 6))
for alpha in alphas:
    if alpha in train_histories and alpha in val_histories:
        plt.plot(train_histories[alpha], label=f'Train α={alpha}', linestyle='-')
        plt.plot(val_histories[alpha], label=f'Val α={alpha}', linestyle='--')

plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.title('Training vs Validation Cost per Alpha (Cost Function 1)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

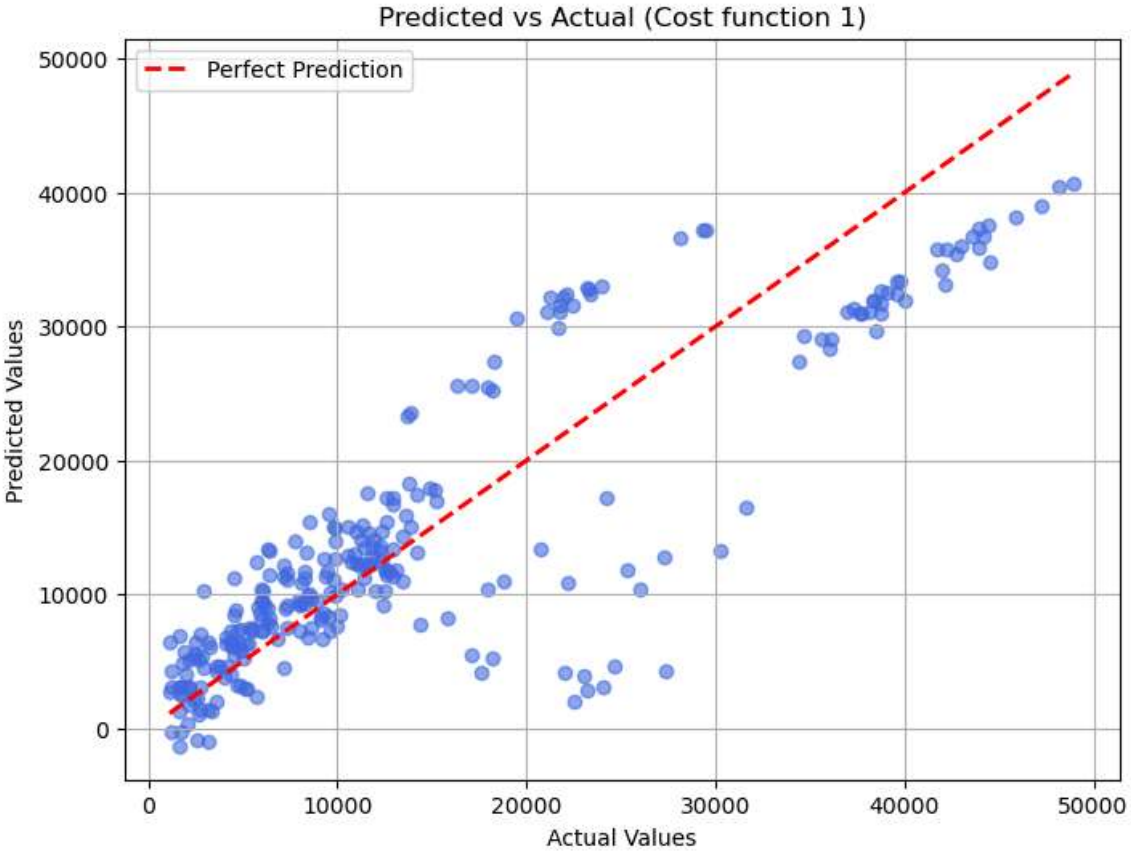
```
best_val_cost1 = 4485495717.284618
best_alpha1 = 0.001
best_theta1 = [[13258.2279861 ]
 [ 3690.85666312]
 [ 2054.83579061]
 [ 9467.99237042]]
```



In [41]:
```
# this code generated by chatgpt for "Add Polynomial Features part"; but i chang it to fit it in my Linear regression mod
from sklearn.metrics import mean_squared_error, r2_score
```

```
y_pred_custom = X_test_b.dot(best_theta1)

alpha = best_alpha1

mse = mean_squared_error(y_test, y_pred_custom)
r2 = r2_score(y_test, y_pred_custom)

print("Custom Linear - MSE:", mse, " R2:", r2)

import matplotlib.pyplot as plt

# Flatten predictions and actual values if they are column vectors
y_pred_flat = y_pred_custom.flatten()
y_actual_flat = y_test.flatten()

plt.figure(figsize=(8, 6))
plt.scatter(y_actual_flat, y_pred_flat, alpha=0.6, color='royalblue')
plt.plot([y_actual_flat.min(), y_actual_flat.max()],
         [y_actual_flat.min(), y_actual_flat.max()],
         'r--', linewidth=2, label='Perfect Prediction')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Predicted vs Actual (Cost function 1)')
plt.legend()
plt.grid(True)
plt.show()
```

```
Custom Linear - MSE: 38473355.437042  R2: 0.7519609398649715
```

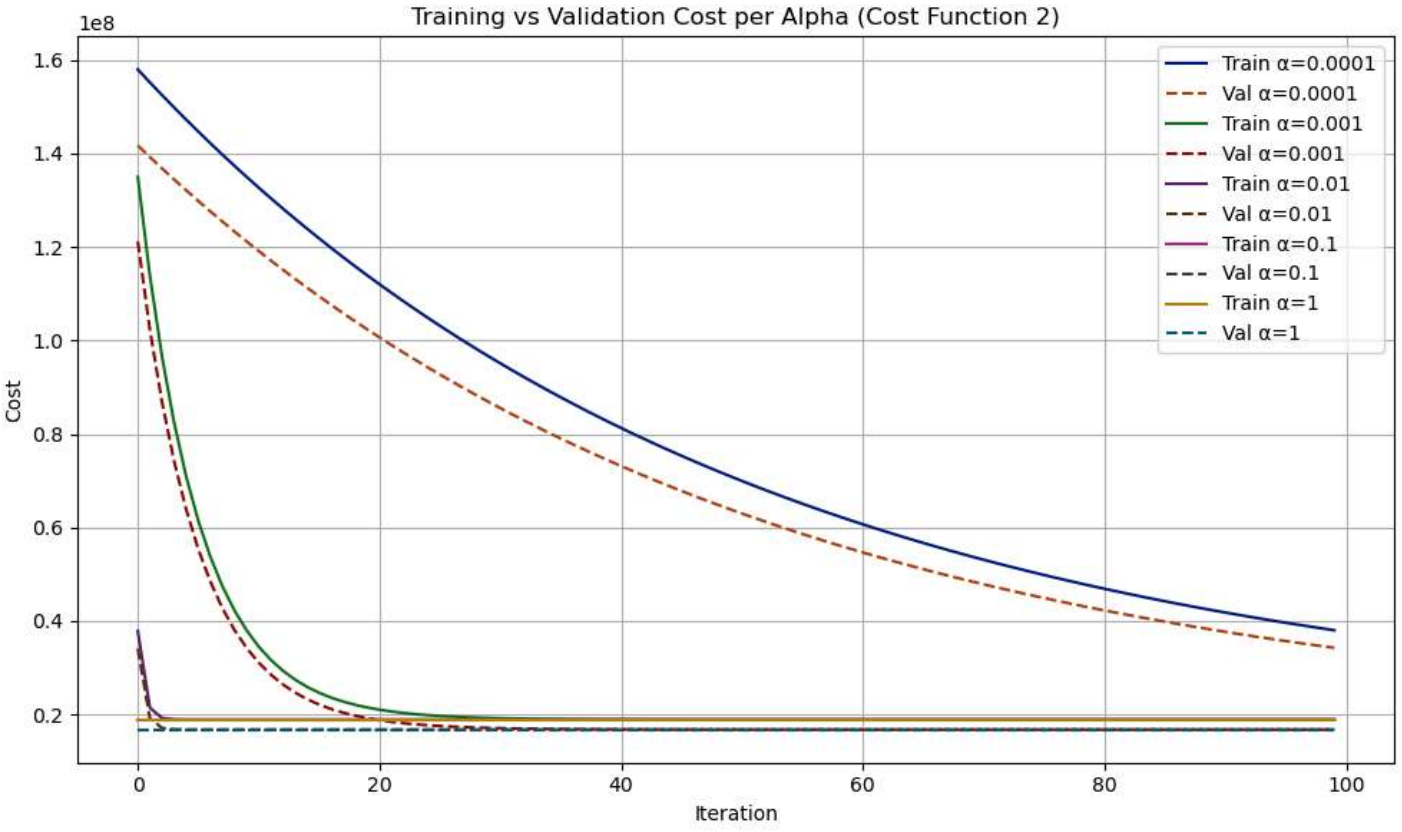


In [42]:

```
# chatgpt generation codes
cost_fn = 2
best_val_cost2 = float('inf')
best_alpha2 = None
best_theta2 = None

train_histories = {}
val_histories = {}

for alpha in alphas:
    theta_init = np.zeros((X_train_b.shape[1], 1))
    theta = theta_init.copy()
    train_costs = []
    val_costs = []

    # Manual training loop to collect validation costs per iteration
    for i in range(iterations):
```

```
        theta, cost_history = gradient_descent(X_train_b, y_train, theta, alpha, iterations,cost_fn=2)# i add this to cal
        # first code do the learning here

        train_cost = cost_function_2(X_train_b, y_train, theta)
        val_cost = cost_function_2(X_val_b, y_val, theta)

        train_costs.append(train_cost)
        val_costs.append(val_cost)

    # Store histories
    train_histories[alpha] = train_costs
    val_histories[alpha] = val_costs

    # Track best model by lowest final val cost
    if len(val_costs) > 0 and val_costs[-1] < best_val_cost2:
        best_val_cost2 = val_costs[-1]
        best_alpha2 = alpha
        best_theta2 = theta

print(f'best_val_cost2 = {best_val_cost2}')
print(f'best_alpha2 = { best_alpha2}')
print(f'best_theta2 = {best_theta2}')

# Plot training vs validation costs
plt.figure(figsize=(10, 6))
for alpha in alphas:
    if alpha in train_histories and alpha in val_histories:
        plt.plot(train_histories[alpha], label=f'Train α={alpha}', linestyle='-')
        plt.plot(val_histories[alpha], label=f'Val α={alpha}', linestyle='--')

plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.title('Training vs Validation Cost per Alpha (Cost Function 2)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

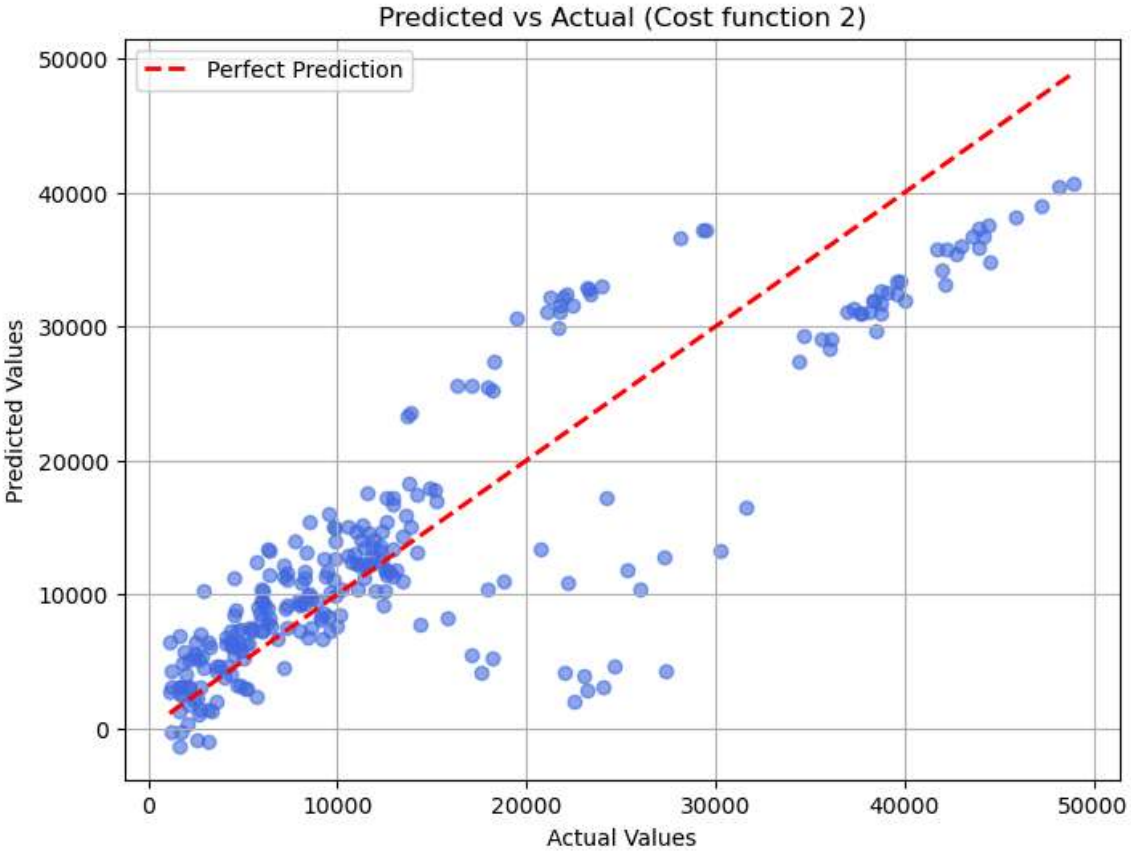
```
best_val_cost2 = 16736924.318226188
best_alpha2 = 0.001
best_theta2 = [[13258.2279861 ]
 [ 3690.85666312]
 [ 2054.83579061]
 [ 9467.99237042]]
```



In [43]:
```
# this code generated by chatgpt for "Add Polynomial Features part"; but i chang it to fit it in my Linear regression mod
from sklearn.metrics import mean_squared_error, r2_score
```

```
y_pred_custom = X_test_b.dot(best_theta2)

alpha = best_alpha2

mse = mean_squared_error(y_test, y_pred_custom)
r2 = r2_score(y_test, y_pred_custom)

print("Custom Linear - MSE:", mse, " R2:", r2)

import matplotlib.pyplot as plt

# Flatten predictions and actual values if they are column vectors
y_pred_flat = y_pred_custom.flatten()
y_actual_flat = y_test.flatten()

plt.figure(figsize=(8, 6))
plt.scatter(y_actual_flat, y_pred_flat, alpha=0.6, color='royalblue')
plt.plot([y_actual_flat.min(), y_actual_flat.max()],
         [y_actual_flat.min(), y_actual_flat.max()],
         'r--', linewidth=2, label='Perfect Prediction')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Predicted vs Actual (Cost function 2)')
plt.legend()
plt.grid(True)
plt.show()
```

```
Custom Linear - MSE: 38473355.437042  R2: 0.7519609398649715
```



Explanations and Discussions

Steps:

1. **Dataset Preprocessing**:
   - Load the dataset and do some statistics and visualizations processing to understand the dataset in depth.
   - Encode any nonnumerical values.
   - ensures the dataset does not have any null values that may affect my results.
   - Extract features `['age', 'bmi', 'smoker_yes']` and labels `['charges']`.
   - Split my dataset: first into train (60%) and temp (40%).
   - Then split temp into validation (20%) and test (20%).
   - Add a bias column to each set.
2. **Training and validation**:

- Start with initial guesses for the model weights (e.g., `theta = 0`).
- **Compute the gradient (vectorized gradient):**
  - *Cost function - 1*: $J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$ => $\nabla_\theta J(\theta) = X^T(X\theta - y)$
    - **Note**: I change the gradient for cost function 1 and divide it by m because it shows an overflow error. I explained that in the comments. Plus, I do some standardization for my dataset, but I still have the same error; I was forced to go with that.
  - *Cost function - 2*: $J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta(x^{(i)})\right)^2$ => $\nabla_\theta J(\theta) = \frac{1}{m}X^T(X\theta - y)$
- **Set a learning rate** – Create a list for all required learning rates α = {0.0001, 0.001, 0.01, 0.1, 1, 10, 100} and loop over each one and train the model and choose the best learning rate for each cost function.
  - **Note**: I discard α = {10, 100} because it shows an overflow error; I already explained that in my comments.

3. **Results**: So, as shown in the plots the best learning rate is 0.001.
   - Custom Linear: *MSE*: 38473355.437042 - *R2*: 0.7519609398649715
     - MSE $\approx$ 38473355.437042 and i think this is normal because my output are charges, so my predictions are off by about $\sqrt{38473355.437042} \approx$ \$6200
   - Also my model has $R^2 \approx 0.75$ so the model explains 75% of the variance in package costs, which is pretty solid, but there's still about 25% of the variance not accounted for.

Custom Model VS. Sklearn Model:

In [46]:
```
# first version of  this code generated by chatgpt ; but after i splitting the dataset to (training, validatin, testing)
# i change it to apply it to X_test set
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score


y_pred_custom_test = X_test_b.dot(best_theta1)

# Train sklearn model
lin_reg = LinearRegression()
lin_reg.fit(X_train_b, y_train)
y_pred_sklearn = lin_reg.predict(X_test_b)

# Evaluation
mse_custom = mean_squared_error(y_test, y_pred_custom_test)
r2_custom = r2_score(y_test, y_pred_custom_test)

mse_sklearn = mean_squared_error(y_test, y_pred_sklearn)
r2_sklearn = r2_score(y_test, y_pred_sklearn)

print("Custom Model - MSE:", mse_custom, " R2:", r2_custom)
print("Sklearn Model - MSE:", mse_sklearn, " R2:", r2_sklearn)
```

```
Custom Model - MSE: 38473355.437042  R2: 0.7519609398649715
Sklearn Model - MSE: 38473014.295888245  R2: 0.7519631392138505
```

Custom Model VS. Sklearn Model Discussions:

- **Custom Model** - MSE: 38473355.437042 R2: 0.7519609398649715
- **Sklearn Model** - MSE: 38473014.295888245 R2: 0.7519631392138505

The performance match, so I think I did a great job :).

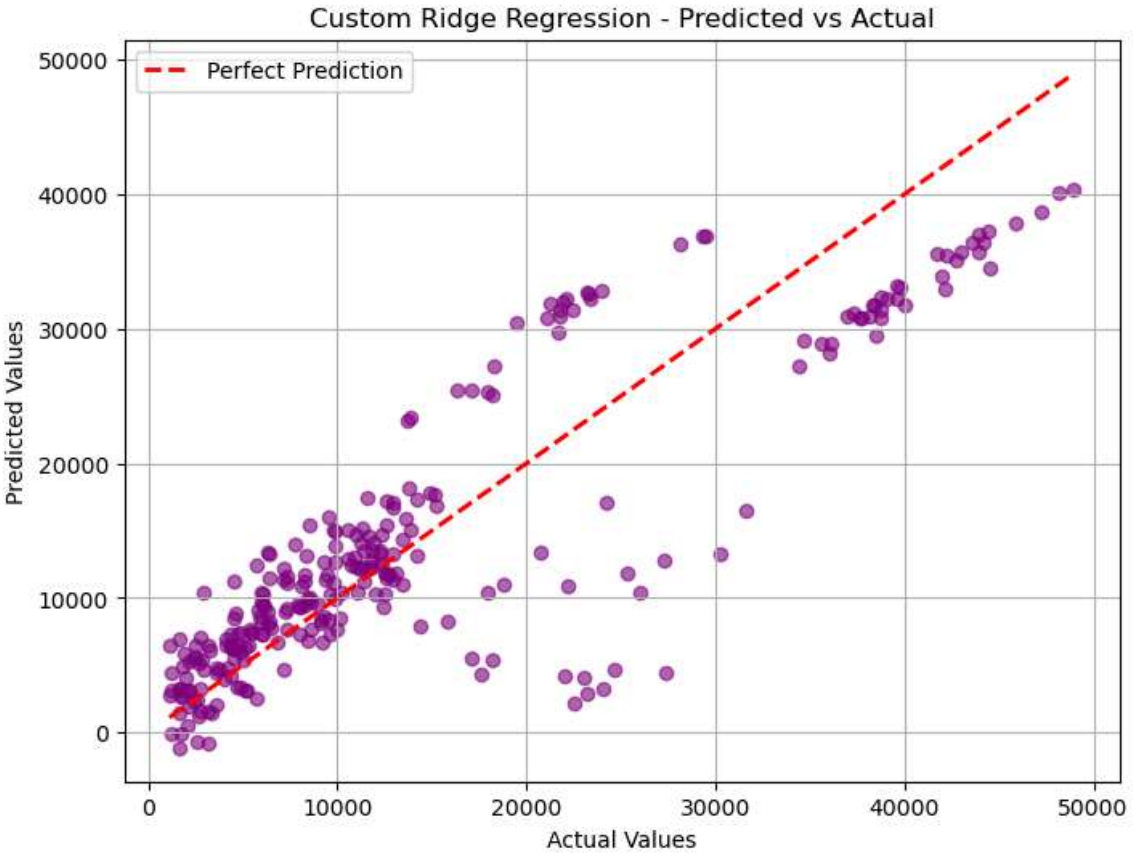
Add Regularization (Ridge)

Modify Gradient Descent to Include Regularization:

In [49]:
```
import numpy as np
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt


# ==== Step 1: Add bias term ====
def add_bias(X):
    return np.hstack([np.ones((X.shape[0], 1)), X])

# ==== Step 2: Custom Ridge Gradient Descent ====
```

```python
def custom_ridge_gradient_descent(X, y, alpha, iterations, lambda_):
    m, n = X.shape
    theta = np.zeros((n, 1))
    cost_history = []

    for _ in range(iterations):
        predictions = X.dot(theta)
        error = predictions - y
        gradient = (1 / m) * X.T.dot(error)
        regularization = (lambda_ / m) * np.r_[[[0]], theta[1:]]  # don't regularize bias
        theta = theta - alpha * (gradient + regularization)

        cost = (1 / (2 * m)) * np.sum(error ** 2) + (lambda_ / (2 * m)) * np.sum(theta[1:] ** 2)
        cost_history.append(cost)

    return theta, cost_history


X_train_b = add_bias(X_train)
X_test_b = add_bias(X_test)

# ==== Step 4: Train Model ====
alpha = 0.001
iterations = 10000
lambda_ = 10

theta_final, cost_history = custom_ridge_gradient_descent(X_train_b, y_train, alpha, iterations, lambda_)

# ==== Step 5: Evaluate ====
y_pred = X_test_b.dot(theta_final)

mse_ridge = mean_squared_error(y_test, y_pred)
r2_ridge = r2_score(y_test, y_pred)

print("Custom Ridge Regression", "MSE:", mse_ridge, "R² Score:", r2_ridge)

# ==== Step 6: Plot Predictions ====
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, alpha=0.6, color='purple')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],
         'r--', linewidth=2, label='Perfect Prediction')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Custom Ridge Regression - Predicted vs Actual')
plt.legend()
plt.grid(True)
plt.show()
```

```
Custom Ridge Regression MSE: 38503135.368020624 R² Score: 0.7517689476145708
```



Custom Linear Model VS. Custom Ridge Model Discussions:

- The difference between my custom linear model and the custom ridge is extremely small, and I think this is because my custom model already generalized well without overfitting.

Compare with sklearn Ridge and Another Model

In [52]:
```
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor

# Ridge from sklearn
ridge = Ridge(alpha=lambda_)
ridge.fit(X_train_b, y_train)
y_pred_ridge_sk = ridge.predict(X_test_b)

mse_ridge_sk = mean_squared_error(y_test, y_pred_ridge_sk)
r2_ridge_sk = r2_score(y_test, y_pred_ridge_sk)

# Try Random Forest (or any model)
rf = RandomForestRegressor(random_state=42)
rf.fit(X_train_b, y_train.ravel())
y_pred_rf = rf.predict(X_test_b)

mse_rf = mean_squared_error(y_test, y_pred_rf)
r2_rf = r2_score(y_test, y_pred_rf)

print("Sklearn Ridge - MSE:", mse_ridge_sk, " R2:", r2_ridge_sk)
print("Random Forest - MSE:", mse_rf, " R2:", r2_rf)
```

```
Sklearn Ridge - MSE: 38502667.44533947  R2: 0.7517719643283648
Random Forest - MSE: 28116791.337559488  R2: 0.8187300686888785
```

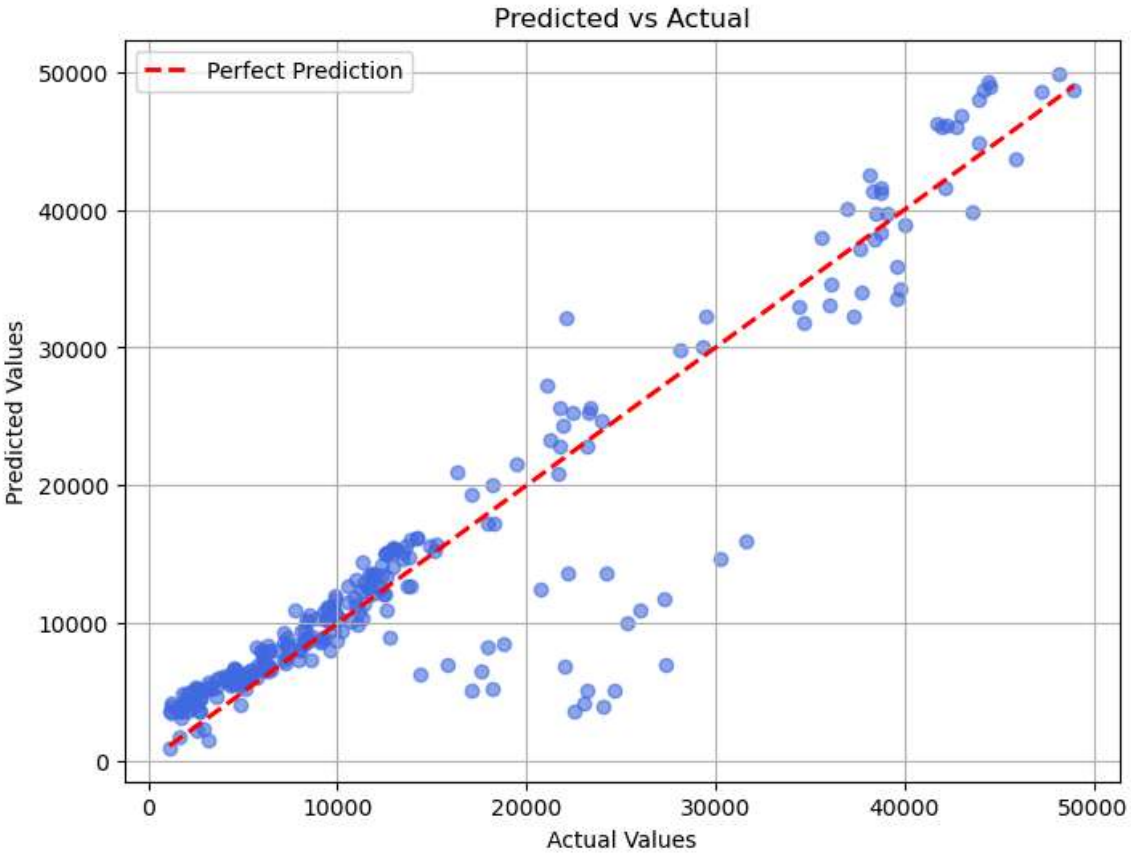
Sklearn Ridge vs. Random Forest Discussions:

- Sklearn Ridge - MSE: 38502667.44533947 R2: 0.7517719643283648
- Random Forest - MSE: 28116791.337559488 R2: 0.8187300686888785
- Random Forest outperforms all my previous models, starting from the first linear custom model until sklearn ridge, and my explanation for that is because the dataset has nonlinear patterns, so Random Forest does capture nonlinear patterns, so it performs very well compared to the other linear models.

Add Polynomial Features

In [55]:
```
import numpy as np
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt

# Step 1: Polynomial feature transformation
poly = PolynomialFeatures(degree=5, include_bias=False)
X_train_poly = poly.fit_transform(X_train_b)
X_test_poly = poly.transform(X_test_b)

# Step 2: Add bias term manually
X_train_b = np.hstack([np.ones((X_train_poly.shape[0], 1)), X_train_poly])
X_test_b = np.hstack([np.ones((X_test_poly.shape[0], 1)), X_test_poly])

# Step 3: Custom gradient descent for polynomial regression
def custom_gradient_descent(X, y, theta, alpha, iterations):
    m = len(y)
    cost_history = []

    for _ in range(iterations):
        predictions = X.dot(theta)
        error = predictions - y
        gradients = (1 / m) * X.T.dot(error)
        theta = theta - alpha * gradients

        cost = (1 / (2 * m)) * np.sum(error ** 2)
        cost_history.append(cost)

    return theta, cost_history

# Step 4: Initialize and run training
alpha = 0.001
iterations = 1000
theta_init = np.zeros((X_train_b.shape[1], 1))

theta_poly_final, cost_history_poly = custom_gradient_descent(
    X_train_b, y_train, theta_init, alpha, iterations
)

# Step 5: Predictions and evaluation
y_pred_poly = X_test_b.dot(theta_poly_final)
mse_poly = mean_squared_error(y_test, y_pred_poly)
r2_poly = r2_score(y_test, y_pred_poly)

print("Custom Polynomial - MSE:", mse_poly, " R²:", r2_poly)

# Step 6: Visualization
y_pred_flat = y_pred_poly.flatten()
y_actual_flat = y_test.flatten()

plt.figure(figsize=(8, 6))
plt.scatter(y_actual_flat, y_pred_flat, alpha=0.6, color='royalblue')
plt.plot([y_actual_flat.min(), y_actual_flat.max()],
         [y_actual_flat.min(), y_actual_flat.max()],
         'r--', linewidth=2, label='Perfect Prediction')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Predicted vs Actual')
plt.legend()
plt.grid(True)
plt.show()
```

```
Custom Polynomial - MSE: 21610884.543371074  R²: 0.8606738759868224
```



In [56]:
```
from sklearn.pipeline import make_pipeline

model_poly_sk = make_pipeline(PolynomialFeatures(degree=2), LinearRegression())
model_poly_sk.fit(X, y)
y_pred_poly_sk = model_poly_sk.predict(X)

mse_poly_sk = mean_squared_error(y, y_pred_poly_sk)
r2_poly_sk = r2_score(y, y_pred_poly_sk)

print("Sklearn Polynomial - MSE:", mse_poly_sk, " R2:", r2_poly_sk)
```

```
Sklearn Polynomial - MSE: 23608879.91702428  R2: 0.8388942662818929
```

Custom Polynomial Model vs. Sklearn Polynomial Model:

As I mentioned above in "Sklearn Ridge vs. Random Forest Discussions," I think it's because the dataset has nonlinear patterns, so the custom polynomial model and Sklearn polynomial model add nonlinearity so the all polynomial model captures nonlinear patterns, so it performs very well compared to the other linear models.

- Custom Polynomial - MSE: 21610884.543371074 R2: 0.8606738759868224
- Sklearn Polynomial - MSE: 23608879.91702428 R2: 0.8388942662818929
- Both models are performing well, but my custom polynomial model outperforms a little bit.
- **Note**: I tried some degrees, and degree 5 gave me the best performance, but if I ran the same code again, it showed an error, so I just restarted the kernel and tried one run to get the best outcomes.

In [58]:
```
import pandas as pd

results = pd.DataFrame({
    'Model': [
        'Custom Linear',
        'Sklearn Linear',
        'Custom Ridge',
        'Sklearn Ridge',
        'Random Forest',
        'Custom Polynomial',
        'Sklearn Polynomial'
    ],
    'MSE': [
        mse_custom,
        mse_sklearn,
        mse_ridge,
        mse_ridge_sk,
        mse_rf,
```

```
        mse_poly,
        mse_poly_sk
    ],
    'R² Score': [
        r2_custom,
        r2_sklearn,
        r2_ridge,
        r2_ridge_sk,
        r2_rf,
        r2_poly,
        r2_poly_sk
    ]
})

print(results)
```

```
              Model           MSE  R² Score
0      Custom Linear  3.847336e+07  0.751961
1     Sklearn Linear  3.847301e+07  0.751963
2       Custom Ridge  3.850314e+07  0.751769
3      Sklearn Ridge  3.850267e+07  0.751772
4      Random Forest  2.811679e+07  0.818730
5  Custom Polynomial  2.161088e+07  0.860674
6  Sklearn Polynomial  2.360888e+07  0.838894
```

Conclusion:

Above, I show my final results, and as we can observe, all models with non-linear capabilities do better than linear ones. So, in my opinion, if I tried to produce a good model for a specific problem, I'd have to know if the dataset has nonlinearity. I had to deal with that by polynomial model or nonlinear model.

And finally, I always prefer nonlinear models because I think almost all data in our real world has a nonlinearity pattern.

References:

- [1] https://www.kaggle.com/code/narminhumbatli/medical-cost-personal-datasets-data-visualization
- [2] https://www.kaggle.com/code/mragpavank/medical-cost-personal-datasets/notebook
- [3] https://www.kaggle.com/datasets/mirichoi0218/insurance
- [4] https://www.geeksforgeeks.org/gradient-descent-in-linear-regression/