In [1]:
```
# We are using combinations of nameless features in a Mercedes-Benz manufacturing process to predict and optimize the v

import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import xgboost as xgb


# Read the train CSV file into a DataFrame
df_train = pd.read_csv('/Users/wkammerait/Desktop/ML Data Sets/Mercedes Data Sets/train mercedes.csv')

df_test = pd.read_csv('/Users/wkammerait/Desktop/ML Data Sets/Mercedes Data Sets/test mercedes.csv')

df_train.head()
```

Out[1]:

	ID	y	X0	X1	X2	X3	X4	X5	X6	X8	...	X375	X376	X377	X378	X379	X380	X382	X383	X384	X385
0	0	130.81	k	v	at	a	d	u	j	o	...	0	0	1	0	0	0	0	0	0	0
1	6	88.53	k	t	av	e	d	y	l	o	...	1	0	0	0	0	0	0	0	0	0
2	7	76.26	az	w	n	c	d	x	j	x	...	0	0	0	0	0	0	1	0	0	0
3	9	80.62	az	t	n	f	d	x	l	e	...	0	0	0	0	0	0	0	0	0	0
4	13	78.02	az	v	n	f	d	h	d	n	...	0	0	0	0	0	0	0	0	0	0

5 rows × 378 columns

In [2]:
```
df_test.head()
```

Out[2]:

	ID	X0	X1	X2	X3	X4	X5	X6	X8	X10	...	X375	X376	X377	X378	X379	X380	X382	X383	X384	X385
0	1	az	v	n	f	d	t	a	w	0	...	0	0	0	1	0	0	0	0	0	0
1	2	t	b	ai	a	d	b	g	y	0	...	0	0	1	0	0	0	0	0	0	0
2	3	az	v	as	f	d	a	j	j	0	...	0	0	0	1	0	0	0	0	0	0
3	4	az	l	n	f	d	z	l	n	0	...	0	0	0	1	0	0	0	0	0	0
4	5	w	s	as	c	d	y	i	m	0	...	1	0	0	0	0	0	0	0	0	0

5 rows × 377 columns

In [3]:
```
# We could drop 0-variance columns here. Instead we are dropping low-variance columns.

# Calculate the variance for each column
variances = df_train.var()

# Get the column names with zero variance
zero_variance_columns = variances[variances == 0].index.tolist()

# Drop columns with zero variance -- drop these from test data as well.
df_train = df_train.drop(columns=zero_variance_columns)
df_test = df_test.drop(columns=zero_variance_columns)

# Display the dropped columns
print("Dropped columns with == 0 variance:")
print(zero_variance_columns)
```

```
Dropped columns with == 0 variance:
['X11', 'X93', 'X107', 'X233', 'X235', 'X268', 'X289', 'X290', 'X293', 'X297', 'X330', 'X347']

/var/folders/5d/4jt8vby95mg87766vj65w20c0000gn/T/ipykernel_17822/168799594.py:4: FutureWarning: Dropping of nuisance columns in DataFrame reductions (with 'numeric_only=None') is deprecated; in a future version this will raise TypeErr or.  Select only valid columns before calling the reduction.
  variances = df_train.var()
```

In [5]:
```
# We can double check 0 variances here if necessary.
variances
```

Out[5]:
```
ID      5.941936e+06
y       1.607667e+02
X10     1.313092e-02
X11     0.000000e+00
X12     6.945713e-02
            ...
X380    8.014579e-03
X382    7.546747e-03
X383    1.660732e-03
X384    4.750593e-04
X385    1.423823e-03
Length: 370, dtype: float64
```

In [6]:
```
# Calculate the variance for each column
variances = df_train.var()

# Reset index and convert to DataFrame
variances_df = variances.reset_index()

# Rename the columns of the DataFrame
variances_df.columns = ['Column', 'Variance']

# Display the column names and variances
print(variances_df)
```

```
    Column      Variance
0       ID  5.941936e+06
1        y  1.607667e+02
2      X10  1.313092e-02
3      X12  6.945713e-02
4      X13  5.462335e-02
..     ...           ...
353   X380  8.014579e-03
354   X382  7.546747e-03
355   X383  1.660732e-03
356   X384  4.750593e-04
357   X385  1.423823e-03

[358 rows x 2 columns]

/var/folders/5d/4jt8vby95mg87766vj65w20c0000gn/T/ipykernel_17822/3912459915.py:2: FutureWarning: Dropping of nuisance
columns in DataFrame reductions (with 'numeric_only=None') is deprecated; in a future version this will raise TypeErr
or.  Select only valid columns before calling the reduction.
  variances = df_train.var()
```

In [7]:
```
# We see that there are no null values in the training data.
null_counts = df_train.isnull().sum()

sorted_null_counts = null_counts.sort_values(ascending=False)

print(sorted_null_counts.head(40))
```

```
ID      0
X262    0
X260    0
X259    0
X258    0
X257    0
X256    0
X255    0
X254    0
X253    0
X252    0
X251    0
X250    0
X249    0
X248    0
X247    0
X246    0
X245    0
X244    0
X243    0
X242    0
X261    0
X263    0
X240    0
X264    0
X284    0
X283    0
X282    0
X281    0
X280    0
X279    0
X278    0
X277    0
X276    0
X275    0
X274    0
X273    0
X272    0
X271    0
X270    0
dtype: int64
```

In [8]:
```
# We see that there are no null values in the test data.
null_counts = df_test.isnull().sum()

sorted_null_counts = null_counts.sort_values(ascending=False)

print(sorted_null_counts.head(40))
```

```
ID      0
X263    0
X261    0
X260    0
X259    0
X258    0
X257    0
X256    0
X255    0
X254    0
X253    0
X252    0
X251    0
X250    0
X249    0
X248    0
X247    0
X246    0
X245    0
X244    0
X243    0
X262    0
X264    0
X241    0
X265    0
X285    0
X284    0
X283    0
X282    0
X281    0
X280    0
X279    0
X278    0
X277    0
X276    0
X275    0
X274    0
X273    0
X272    0
X271    0
dtype: int64
```

In [9]:
```
# Count unique values in the training data.
unique_counts = df_train.nunique()

# Sort the unique counts in descending order
sorted_unique_counts = unique_counts.sort_values(ascending=False)

# Print the unique counts
print(sorted_unique_counts)
```

```
ID      4209
y       2545
X0        47
X2        44
X5        29
        ... 
X131       2
X130       2
X129       2
X128       2
X385       2
Length: 366, dtype: int64
```

In [10]:
```
# Count unique values in the training data.
unique_counts = df_test.nunique()

# Sort the unique counts in descending order
sorted_unique_counts = unique_counts.sort_values(ascending=False)

# Print the unique counts
print(sorted_unique_counts)
```

```
ID      4209
X0        49
X2        45
X5        32
X1        27
        ...
X369       1
X257       1
X258       1
X296       1
X295       1
Length: 365, dtype: int64
```

In [11]:
```
# Apply label encoder.
class CustomEncoder:
    def __init__(self):
        self.mapping = {}

    def fit_transform(self, data):
        codes, uniques = pd.factorize(data)
        self.mapping = {unique: code for code, unique in enumerate(uniques)}
        return codes

    def transform(self, data):
        return [self.mapping.get(x, -1) for x in data]

encoders = {}
rows_with_new_values = set()
common_columns = set(df_train.columns) & set(df_test.columns)

for column in df_train.columns:
    if column in common_columns and df_train[column].dtype == 'object':
        encoders[column] = CustomEncoder()
        df_train[column] = encoders[column].fit_transform(df_train[column])

for column in df_test.columns:
    if column in common_columns and df_test[column].dtype == 'object':
        transformed_data = encoders[column].transform(df_test[column])
        rows_with_new_values.update(i for i, v in enumerate(transformed_data) if v == -1)
        df_test[column] = transformed_data

print(f"Number of unique rows in the test set with new values: {len(rows_with_new_values)}")
```

```
Number of unique rows in the test set with new values: 24
```

In [12]:
```
# Check to ensure there are no non-numeric columns remaining
non_numeric_columns = df_test.select_dtypes(exclude='number').columns
print(non_numeric_columns)

df_train.head()
```

```
Index([], dtype='object')
```

Out[12]:

	ID	y	X0	X1	X2	X3	X4	X5	X6	X8	...	X375	X376	X377	X378	X379	X380	X382	X383	X384	X385
0	0	130.81	0	0	0	0	0	0	0	0	...	0	0	1	0	0	0	0	0	0	0
1	6	88.53	0	1	1	1	0	1	1	0	...	1	0	0	0	0	0	0	0	0	0
2	7	76.26	1	2	2	2	0	2	0	1	...	0	0	0	0	0	0	1	0	0	0
3	9	80.62	1	1	2	3	0	2	1	2	...	0	0	0	0	0	0	0	0	0	0
4	13	78.02	1	0	2	3	0	3	2	3	...	0	0	0	0	0	0	0	0	0	0

5 rows × 366 columns

In [13]:
```
df_test.head()
```

Out[13]:

	ID	X0	X1	X2	X3	X4	X5	X6	X8	X10	...	X375	X376	X377	X378	X379	X380	X382	X383	X384	X385
0	1	1	0	2	3	0	-1	5	16	0	...	0	0	0	1	0	0	0	0	0	0
1	2	2	3	7	0	0	-1	6	18	0	...	0	0	1	0	0	0	0	0	0	0
2	3	1	0	4	3	0	-1	0	13	0	...	0	0	0	1	0	0	0	0	0	0
3	4	1	5	2	3	0	-1	1	3	0	...	0	0	0	1	0	0	0	0	0	0
4	5	5	6	4	2	0	1	4	8	0	...	1	0	0	0	0	0	0	0	0	0

5 rows × 365 columns

In [14]:
```
columns_only_in_test = set(df_test.columns) - set(df_train.columns)
num_columns_only_in_test = len(columns_only_in_test)
print("Number of columns in df_test not present in df_train:", num_columns_only_in_test)
```

```
Number of columns in df_test not present in df_train: 0
```

In [15]:
```
# Perform dimensionality reduction, first determining the minimum number of components to explain 90% of the variance i
import pandas as pd
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Identify the numerical columns in the DataFrame
numerical_columns = df_train.select_dtypes(include=['float64', 'int64']).columns
numerical_columns = numerical_columns.drop('y')  # Exclude 'y' column

# Create a StandardScaler instance
scaler = StandardScaler()

# Fit the scaler to the numerical columns and transform them
df_train[numerical_columns] = scaler.fit_transform(df_train[numerical_columns])
df_test[numerical_columns] = scaler.transform(df_test[numerical_columns])

# Separate the features from the target variable
X_train = df_train.drop(columns=['y'])

# Apply PCA
pca = PCA()

# Fit the PCA to the training data and transform it
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(df_test)

# Calculate cumulative explained variance ratio
explained_variance_ratio_cumulative = np.cumsum(pca.explained_variance_ratio_)

# Find the minimum number of components for desired explained variance ratio
desired_variance_ratio = 0.90  # 90% explained variance
min_components = np.argmax(explained_variance_ratio_cumulative >= desired_variance_ratio) + 1

# Print the minimum number of components
print("Minimum number of components for {} explained variance ratio: {}".format(
    desired_variance_ratio, min_components))
```

```
Minimum number of components for 0.9 explained variance ratio: 120
```

In [16]:
```
# Perform principal component analysis.

import pandas as pd
from sklearn.decomposition import PCA

# Separate the features from the target variable
X = df_train.drop(columns=['y'])

# Apply PCA with 120 components
n_components = 120
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X)

# Print the transformed data
print(X_pca)
```

```
[[ 1.22782184e+01 -2.11063630e+00 -1.03293721e+00 ...  4.97641371e-01
   2.18305662e-01 -1.60180943e-01]
 [-1.86763093e-01  2.57030028e-01  7.93994806e-01 ... -8.91200027e-01
   8.28036275e-01 -1.50310828e+00]
 [ 9.14372781e+00  2.19004460e+01 -3.55807022e+00 ... -3.90013651e+00
   2.46108406e+00  4.87376951e+00]
 ...
 [ 7.66888936e-03  5.63696805e-01  3.32791185e+00 ...  1.60384137e-01
  -1.33603228e-01  1.08401345e-01]
 [-1.52479525e+00  3.62995732e-01 -4.17234655e-01 ... -7.26847538e-01
  -1.77197908e-01  1.27089310e-01]
 [-1.86978202e+00 -1.15907253e+00 -4.21987858e-01 ... -7.16540063e-01
  -1.57363099e+00  4.72781243e-01]]
```

In [18]:
```
#Run the XGBoost model. This is the block used to create the predicted 'y' values for the test data frame. The other bl

from sklearn.model_selection import cross_val_score
from sklearn.metrics import r2_score


# Separate the features from the target variable
X = df_train.drop(columns=['y'])
y = df_train['y']

# Apply PCA with 10 components
pca = PCA(n_components=10)
X_pca = pca.fit_transform(X)

# Split the training data into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_pca, y, test_size=0.2, random_state=42)

# Set default hyperparameter values
model = xgb.XGBRegressor(
    n_estimators=100,
    max_depth=5,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=1.0
)

# Train the XGBoost model on the training data with cross-validation
scores = cross_val_score(model, X_train, y_train, cv=5, scoring='r2')
print("Cross-Validation R-squared Scores:", scores)
print("Mean Cross-Validation R-squared:", scores.mean())

# Fit the model on the training data
model.fit(X_train, y_train)

# Evaluate the model on the training data
predictions_train = model.predict(X_train)

# Calculate R-squared on the training data
r2_train = r2_score(y_train, predictions_train)
print("R-squared on Training Data:", r2_train)

# Evaluate the model on the validation data
predictions_val = model.predict(X_val)

# Calculate R-squared on the validation data
r2_val = r2_score(y_val, predictions_val)
print("R-squared on Validation Data:", r2_val)
```

```
Cross-Validation R-squared Scores: [0.30157902 0.42800399 0.47547081 0.48832433 0.43168915]
Mean Cross-Validation R-squared: 0.4250134627532926
R-squared on Training Data: 0.7181814793904129
R-squared on Validation Data: 0.4367264633549781
```

In [40]:
```
#Run GridSearch to determine optimal hyperparameters.

from sklearn.model_selection import GridSearchCV
import numpy as np

# Create a dictionary of hyperparameters and their ranges
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [5, 50, 500],
    'learning_rate': [0.1, 0.01,1],
    'subsample': [0.6, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0]
}

# Create the XGBRegressor model
model = xgb.XGBRegressor()

# Perform grid search with cross-validation
grid_search = GridSearchCV(model, param_grid, cv=5, scoring='r2')
grid_search.fit(X_train, y_train)

# Get the best hyperparameters and the corresponding mean cross-validated score
best_params = grid_search.best_params_
best_score = grid_search.best_score_

# Print the best hyperparameters and the corresponding score
print("Best Hyperparameters:", best_params)
print("Best R-squared Score:", best_score)
```

```
Best Hyperparameters: {'colsample_bytree': 1.0, 'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 100, 'subsampl
e': 0.8}
Best R-squared Score: 0.4264815821908671
```

In [28]:
```
# This block is using L1 regularization.
model = xgb.XGBRegressor(
    reg_alpha=10,
    learning_rate=0.05,
    subsample=0.8,
    gamma=200,
    n_estimators=100,
    max_depth=15,
    colsample_bytree=1.0)
model.fit(X_train, y_train)

# Evaluate the model on the training data
predictions_train = model.predict(X_train)
mse_train = mean_squared_error(y_train, predictions_train)
r2_train = r2_score(y_train, predictions_train)

# Evaluate the model on the validation data
predictions_val = model.predict(X_val)
mse_val = mean_squared_error(y_val, predictions_val)
r2_val = r2_score(y_val, predictions_val)

print("Mean Squared Error on Training Data:", mse_train)
print("R-squared on Training Data:", r2_train)
print("Mean Squared Error on Validation Data:", mse_val)
print("R-squared on Validation Data:", r2_val)

from sklearn.model_selection import cross_val_score

# Perform cross-validation
cv_scores = cross_val_score(model, X, y, cv=8, scoring='r2')

# Print the cross-validation scores
print("\nCross-validation scores:", cv_scores)
print("Mean R-squared:", cv_scores.mean())
print("Standard Deviation:", cv_scores.std())
```

```
Mean Squared Error on Training Data: 35.52719067568685
R-squared on Training Data: 0.7806925326566663
Mean Squared Error on Validation Data: 88.18357752667934
R-squared on Validation Data: 0.4334493304134476

Cross-validation scores: [0.56980619 0.38179286 0.537391   0.59465469 0.54830418 0.50160005
 0.61275812 0.66842903]
Mean R-squared: 0.5518420136493943
Standard Deviation: 0.07993846100997203
```

In [32]:
```
# Try L2 regularization to see if the performance improves. Using hyperparameter tuning and L2 regularization this mode
model = xgb.XGBRegressor(
    reg_lambda=10,
    n_estimators=100,
    max_depth=15,
    learning_rate=0.05,
    subsample=0.8,
    gamma = 200,
    colsample_bytree=1.0,
    )  # experiment with different values for lambda
model.fit(X_train, y_train)

# Evaluate the model on the validation data
predictions = model.predict(X_val)

# Calculate R-squared score
r2_score_val = r2_score(y_val, predictions)
print("R-squared on Validation Data:", r2_score_val)

# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(y_val, predictions)
print("Mean Squared Error on Validation Data:", mse)

# Perform cross-validation
cv_scores = cross_val_score(model, X, y, cv=8, scoring='r2')

# Print the cross-validation scores
print("\nCross-validation scores:", cv_scores)
print("Mean R-squared:", cv_scores.mean())
print("Standard Deviation:", cv_scores.std())
```

```
R-squared on Validation Data: 0.4650558400096456
Mean Squared Error on Validation Data: 83.26402621565883

Cross-validation scores: [0.58758209 0.3924025  0.5417948  0.65096984 0.54996577 0.50917777
 0.63653682 0.67684235]
Mean R-squared: 0.5681589919218109
Standard Deviation: 0.0859634275915423
```

In [52]:
```
# L2 regularization provided the strongest r^2 score, so these are the values we will use for the predicted targets.
# Separate the features from the target variable in the test set
X_test = df_test

# Apply the same PCA transformation as in the training set
X_test_pca = pca.transform(X_test)

# Predict the target variable for the test set
predictions_test = model.predict(X_test_pca)

# Create a DataFrame with the predicted target values
df_predictions_test = pd.DataFrame({'y': predictions_test})

# Save the predictions to a CSV file
df_predictions_test.to_csv('predictions.csv', index=False)

# Print the DataFrame with the predicted target values
print(df_predictions_test)
```

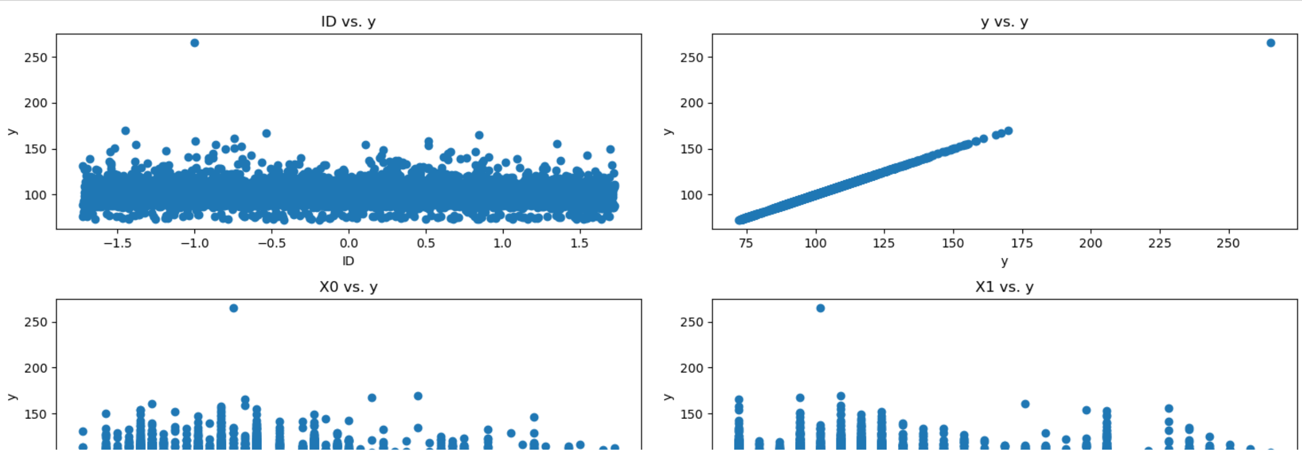
```
               y
0      79.650467
1      97.978767
2      78.057999
3      76.403084
4     111.419075
...          ...
4204  103.983505
4205  100.592323
4206   97.541931
4207  106.358864
4208   95.124207

[4209 rows x 1 columns]
```

In [ ]:
```
# If you need to save these predictions to a CSV file, you can do this:
pd.DataFrame(predictions, columns=['Predicted_y']).to_csv('predictions.csv', index=False)
```

In [29]:
```python
# Run a linear regression for comparison. We see that the R-squared on the validation data is much lower, so this would
from sklearn.linear_model import LinearRegression

# Initialize the model
linear_model = LinearRegression()

# Fit the model on training data
linear_model.fit(X_train, y_train)

# Make predictions on the validation set
predictions_lr = linear_model.predict(X_val)

from sklearn.metrics import mean_squared_error, r2_score

# Make predictions on the training set
predictions_train = model.predict(X_train)

# Compute R-squared score on the training data
r2_score_train = r2_score(y_train, predictions_train)
print("R-squared on Training Data:", r2_score_train)

# Compute Mean Squared Error (MSE) on the training data
mse_train = mean_squared_error(y_train, predictions_train)
print("Mean Squared Error on Training Data:", mse_train)

# Evaluate the model on the validation data
r2_score_val_lr = r2_score(y_val, predictions_lr)
print("R-squared on Validation Data (Linear Regression):", r2_score_val_lr)

# Calculate Mean Squared Error (MSE)
mse_lr = mean_squared_error(y_val, predictions_lr)
print("Mean Squared Error on Validation Data (Linear Regression):", mse_lr)



# Normalization is not included here because it did not materially change the R-squared on the validation data.
```

```
R-squared on Training Data: 0.7806925326566663
Mean Squared Error on Training Data: 35.52719067568685
R-squared on Validation Data (Linear Regression): 0.33660306718859023
Mean Squared Error on Validation Data (Linear Regression): 103.25769255242055
```

In [46]:
```python
# Appendix -- view distributions of each column.

import matplotlib.pyplot as plt

# Assuming selected_columns is your list of the 20 column names
selected_columns = df_train.columns[:20]  # Replace this line with your actual columns

fig, axs = plt.subplots(10, 2, figsize=(15, 30))  # Adjust the size as needed
axs = axs.ravel()

for i, column in enumerate(selected_columns):
    axs[i].scatter(df_train[column], df_train['y'])
    axs[i].set_title(f'{column} vs. y')
    axs[i].set_xlabel(column)
    axs[i].set_ylabel('y')

plt.tight_layout()
plt.show()
```



In [ ]:

In [ ]:

In [ ]: