ECE 469/ECE 568 Machine Learning

Textbook:
Machine Learning: a Probabilistic Perspective by Kevin Patrick Murphy

Southern Illinois University

September 18, 2024

Implementation of Linear Regression in Scikit-Learn

This tutorial will guide you how to implement linear regression in Scikit-learn.
We'll first use the following equation to create the datasets.

$$y = -3 + 5x + n \tag{1}$$

where n is Gaussian noise with 0 mean and σ^2 variance. We'll generate datasets for three variance values ($\sigma^2 \in [1, 5, 20]$).

1. Importing packages

```
import numpy as np
import pandas as pd
```

2. Defining variance values and creating the x values array.

```
variance_values = [1, 5, 20]
x = np.arange(-20, 20.01, 0.01)
```

Implementation of Linear Regression in Scikit-Learn

3. Loop through each variance value to create and save the dataset.

```
# Generating Gaussian noise
n = np.random.normal(0, np.sqrt(variance), x.shape)

# The second arguement in np.random.normal function
requires the standard deviation. Therefore, the standard
deviation should be the input.

# Calculate y using the equation y = -3 + 5x + n
y = -3 + 5 * x + n

# Create a DataFrame
data = pd.DataFrame({
'x': x,
'y': y
})
```

5. Save the data frame to a csv file

```
filename = f'dataset_variance_{variance}.csv'
data.to_csv(filename, index=False)

print(f'Dataset saved to {filename}')
```

6. The final output shown in the kernel is as follows.

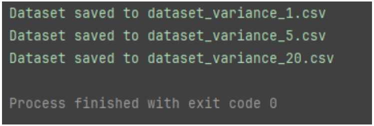


Figure: Output of data set generation

Implementation of Linear Regression in Scikit-Learn

After generating the data sets we'll now implement the linear regression algorithm.

1. Importing packages

```
import pandas as pd
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
```

2. In this task we'll use scikit-learn pacakge to implement linear regression.

Implementation of Linear Regression in Scikit-Learn

1. Loop through each sigma value, load the dataset, and perform stochastic gradient descent SGD regression for variance values.

```
# List of sigma values corresponding to the datasets
variance_values = [1, 5, 20]

# Load the dataset
filename = f'dataset_variance_{variance}.csv'
data = pd.read_csv(filename)

# Extract x and y
x = data['x'].values
y = data['y'].values

# Reshape x for scikit-learn (it expects a 2D array
# for the features)
x = x.reshape(-1, 1)
```

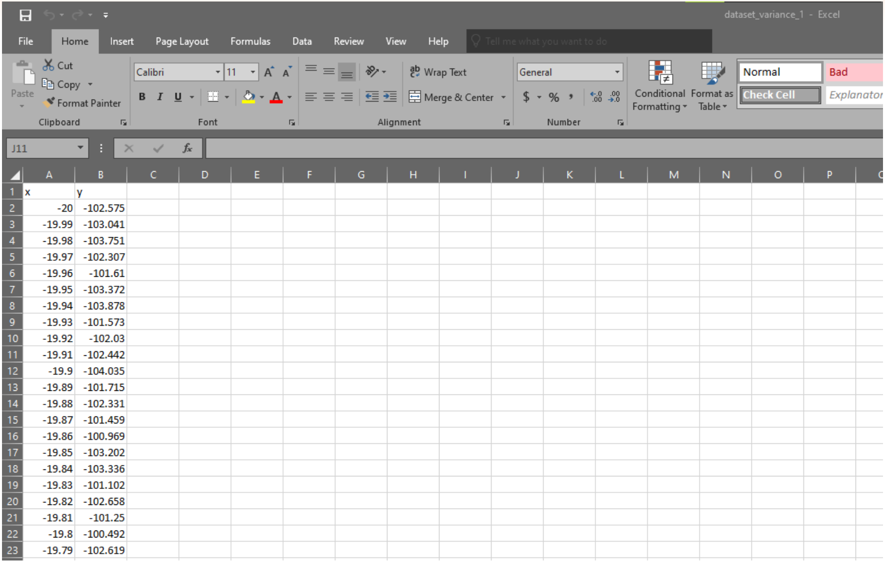
Implementation of Linear Regression in Scikit-Learn

```
# Initialize the SGDRegressor with squared error
# as the loss function
sgd_regressor = SGDRegressor(loss=‘squared_error’,
     max_iter=1000, tol=1e-3, eta0=0.01)

# Fit the model
sgd_regressor.fit(x, y)

# Predict using the trained model
y_pred = sgd_regressor.predict(x)

# Calculate the mean squared error
mse = mean_squared_error(y, y_pred)
```

Implementation of Linear Regression in Scikit-Learn

```
# Print the MSE
print(f"Results for sigma = {variance}:")
print(f"  Mean Squared Error: {mse}\n")

# Plotting the original data and the regression line
plt.figure(figsize=(10, 6))
plt.scatter(x, y, s=2, label='Original Data', alpha=0.6)
plt.plot(x, y_pred, color='red', label='Fitted Line')
plt.title(f'Linear Regression using Gradient Descent
(Variance = {variance})')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

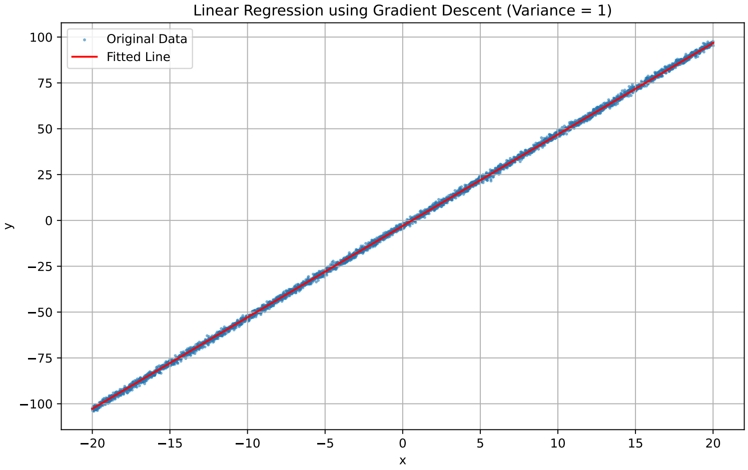
Implementation of Polynomial Regression in Scikit-Learn



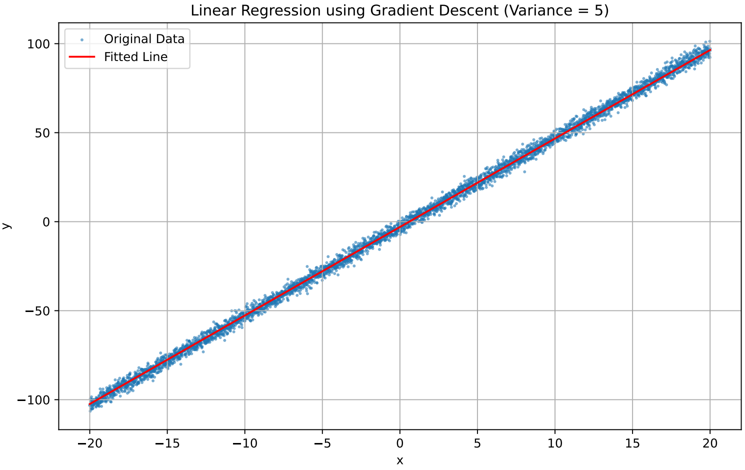
	A	B
1	x	y
2	-20	-102.575
3	-19.99	-103.041
4	-19.98	-103.751
5	-19.97	-102.307
6	-19.96	-101.61
7	-19.95	-103.372
8	-19.94	-103.878
9	-19.93	-101.573
10	-19.92	-102.03
11	-19.91	-102.442
12	-19.9	-104.035
13	-19.89	-101.715
14	-19.88	-102.331
15	-19.87	-101.459
16	-19.86	-100.969
17	-19.85	-103.202
18	-19.84	-103.336
19	-19.83	-101.102
20	-19.82	-102.658
21	-19.81	-101.25
22	-19.8	-100.492
23	-19.79	-102.619

Figure: Dataset for linear regression

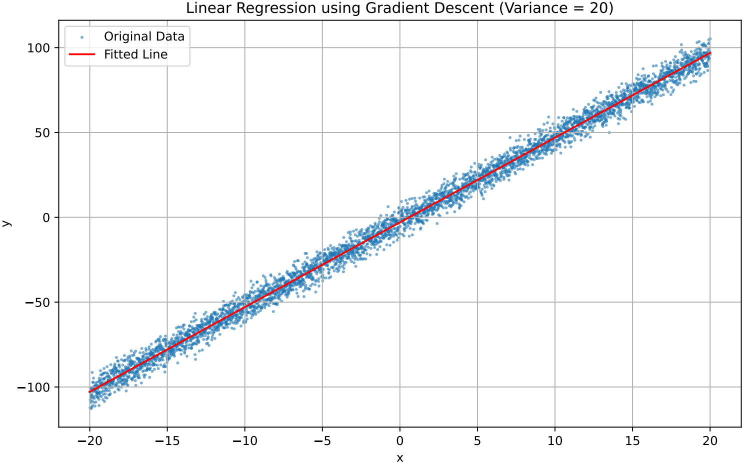
Implementation of Linear Regression in Scikit-Learn



Implementation of Linear Regression in Scikit-Learn



Implementation of Linear Regression in Scikit-Learn



Implementation of Polynomial Regression in Scikit-Learn

This tutorial will guide you how to implement polynomial regression.
In a similar approach to the linear regression task, we create three datasets and implement the polynomial regression using Scikit-Learn.
First we'll create the datasets.

```
import numpy as np
import pandas as pd

# Parameters
variance_values = [1, 5, 20]

# 1000 data samples
n = 1000

# Generate 1000 random x values
x = 6*np.random.rand(n, 1) - 4
```

Implementation of Polynomial Regression in Scikit-Learn

```
# Loop through each sigma value to create and save a dataset
for variance in variance_values:

# Gaussian noise
n = np.random.normal(0, np.sqrt(variance), x.shape)

# Generate y values
y = x ** 2 + 2 * x + 5 + n

# Squeeze the dimension
x1 = np.squeeze(x,1)
y1 = np.squeeze(y,1)

# Create a DataFrame
data = pd.DataFrame({
'x': x1,
'y': y1
})
```

Implementation of Polynomial Regression in Scikit-Learn

```
# Save the DataFrame to a CSV file
filename = f'dataset1_variance_{variance}.csv'
data.to_csv(filename, index=False)

print(f'Dataset saved to {filename}')
```

Implementation of Polynomial Regression in Scikit-Learn

Polynomial regression task:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# List of sigma values corresponding to the datasets
variance_values = [1, 5, 20]
degree = 5  # Degree of the polynomial
test_size = 0.2  # 20% of the data will be used for testing

# Loop through each sigma value, load the dataset,
# and perform polynomial regression
```

Implementation of Polynomial Regression in Scikit-Learn

Polynomial regression task:

```
# Load the dataset
filename = f'dataset1_variance_{variance}.csv'
data = pd.read_csv(filename)

# Extract x and y
x = data['x'].values
y = data['y'].values

# Reshape x for scikit-learn (it expects a 2D array for
# the features)
x = x.reshape(-1, 1)

# Split the data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(x, y,
 test_size=test_size, random_state=42)
```

Implementation of Polynomial Regression in Scikit-Learn

Polynomial regression task:

```
# Transform the original x data into polynomial features
poly = PolynomialFeatures(degree = degree)
x_train_poly = poly.fit_transform(x_train)
x_test_poly = poly.transform(x_test)

# Initialize and fit the linear regression model on the
# training data
poly_regressor = LinearRegression()
poly_regressor.fit(x_train_poly, y_train)

# Predict using the trained model on both
# training and testing data
y_train_pred = poly_regressor.predict(x_train_poly)
y_test_pred = poly_regressor.predict(x_test_poly)
```

Implementation of Polynomial Regression in Scikit-Learn

Polynomial regression task:

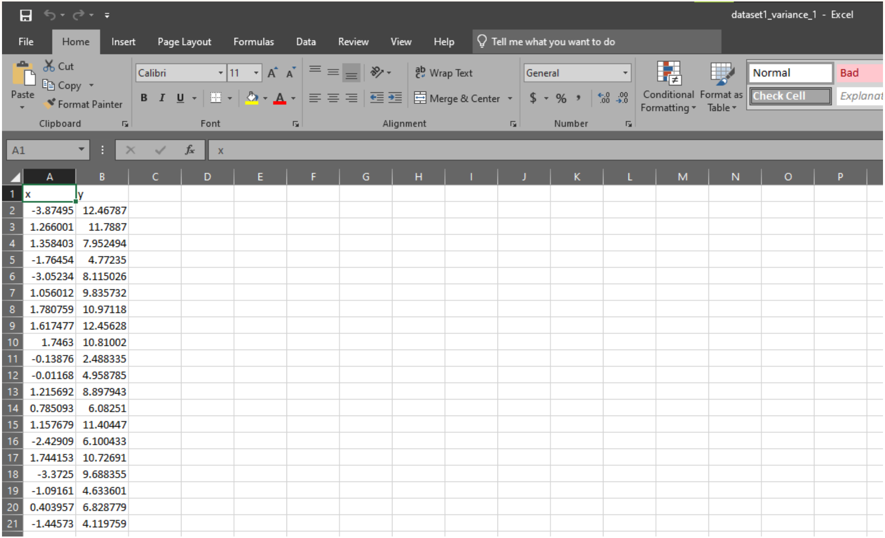
```
# Calculate the mean squared error for both
# training and testing sets
mse_train = mean_squared_error(y_train, y_train_pred)
mse_test = mean_squared_error(y_test, y_test_pred)

# Print the MSE for both training and testing sets
print(f"Results for variance = {variance}:")
print(f"  Mean Squared Error (Training Set): {mse_train}")
print(f"  Mean Squared Error (Testing Set): {mse_test}\n")
```

Implementation of Polynomial Regression in Scikit-Learn

Polynomial regression task:

```
# Plotting the results
plt.figure(figsize=(10, 6))
plt.scatter(x_train, y_train, s=10, label=‘Training Data’,
alpha=0.6)
plt.scatter(x_test, y_test, s=10, color=‘orange’,
label=‘Testing Data’, alpha=0.6)

plt.plot(np.sort(x_train, axis=0),
poly_regressor.predict(poly.transform(np.sort(x_train, axis=0))), c
plt.title(f‘Polynomial Regression (Degree {degree}) using
variance = {variance}’)

plt.xlabel(‘x’)
plt.ylabel(‘y’)
plt.legend()
plt.grid(True)
plt.show()
```

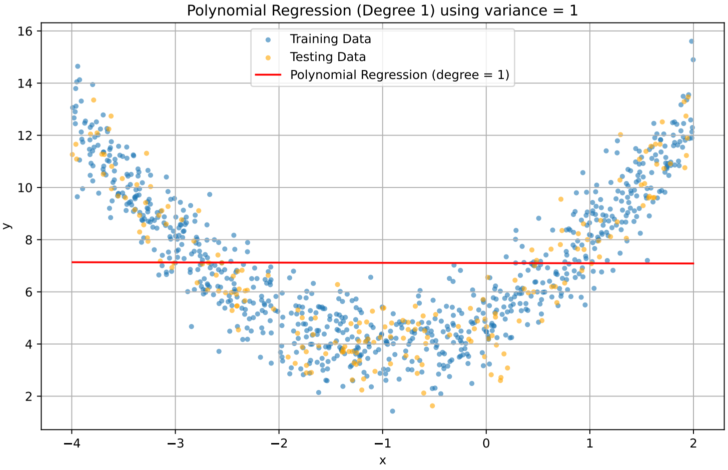
Implementation of Polynomial Regression in Scikit-Learn



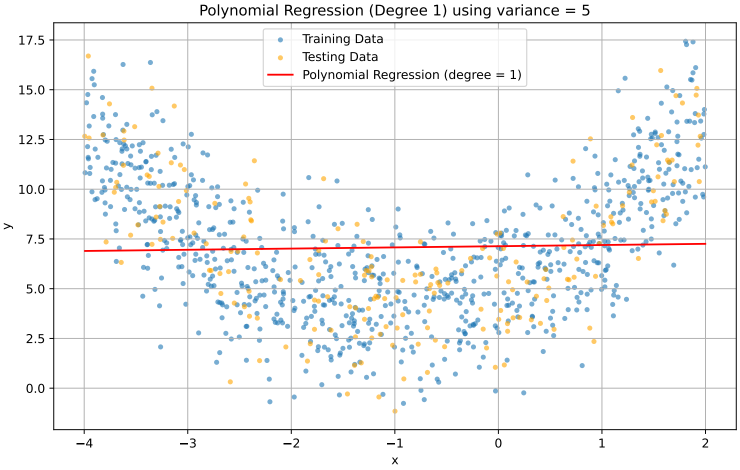
x	y
-3.87495	12.46787
1.266001	11.7887
1.358403	7.952494
-1.76454	4.77235
-3.05234	8.115026
1.056012	9.835732
1.780759	10.97118
1.617477	12.45628
1.7463	10.81002
-0.13876	2.488335
-0.01168	4.958785
1.215692	8.897943
0.785093	6.08251
1.157679	11.40447
-2.42909	6.100433
1.744153	10.72691
-3.3725	9.688355
-1.09161	4.633601
0.403957	6.828779
-1.44573	4.119759

Figure: Dataset for polynomial regression

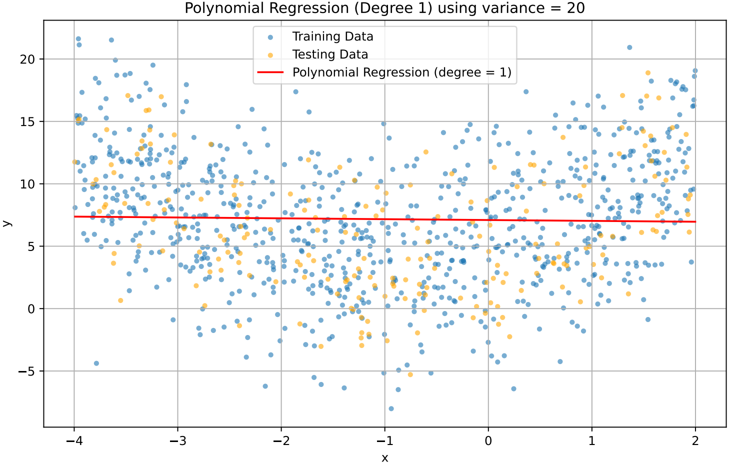
Implementation of Polynomial Regression in Scikit-Learn



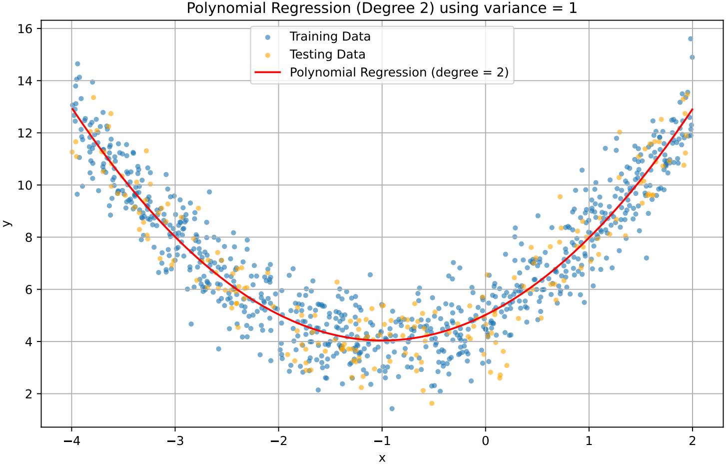
Implementation of Polynomial Regression in Scikit-Learn



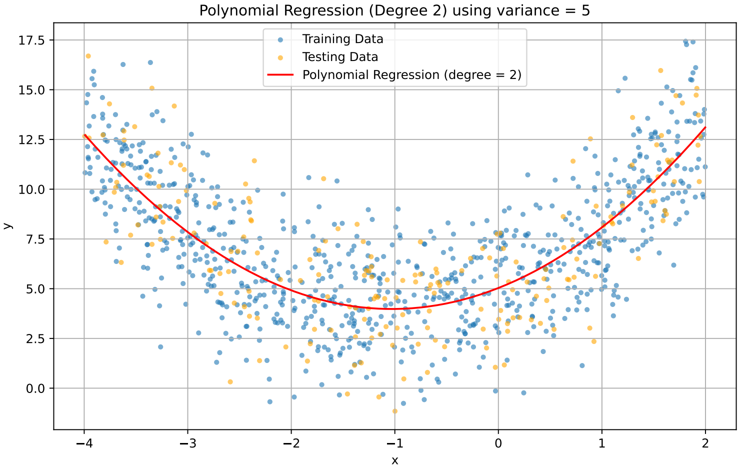
Implementation of Polynomial Regression in Scikit-Learn



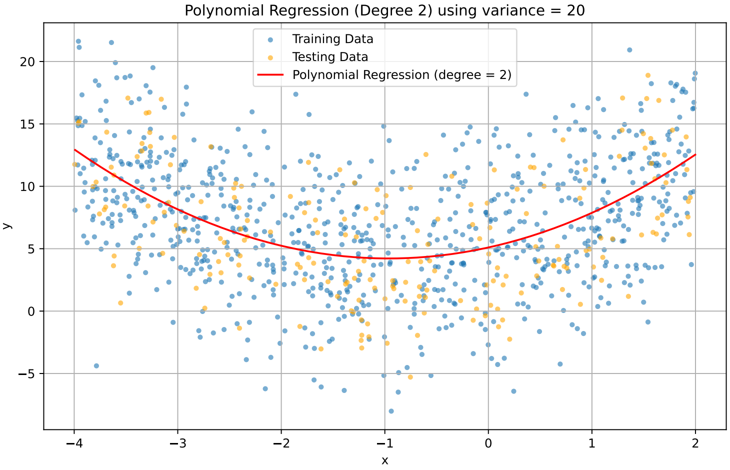
Implementation of Polynomial Regression in Scikit-Learn



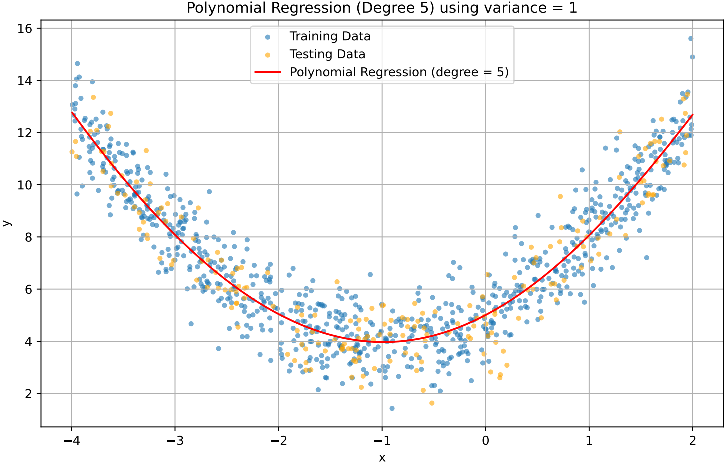
Implementation of Polynomial Regression in Scikit-Learn



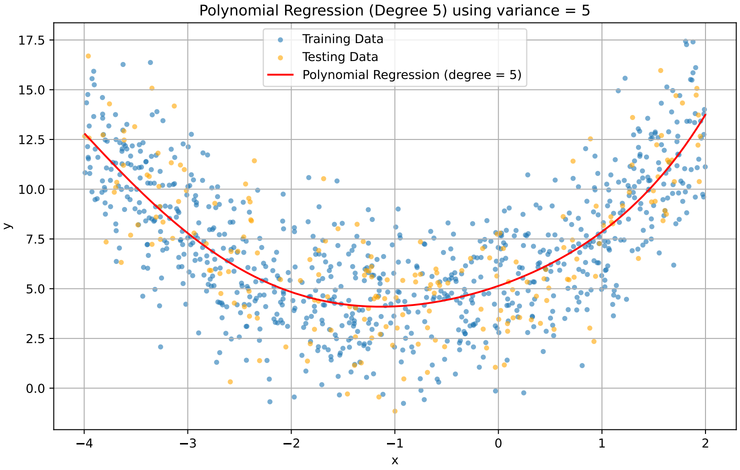
Implementation of Polynomial Regression in Scikit-Learn



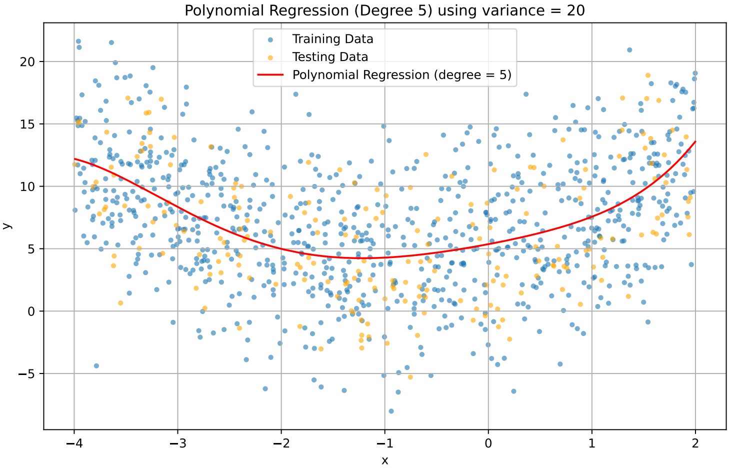
Implementation of Polynomial Regression in Scikit-Learn



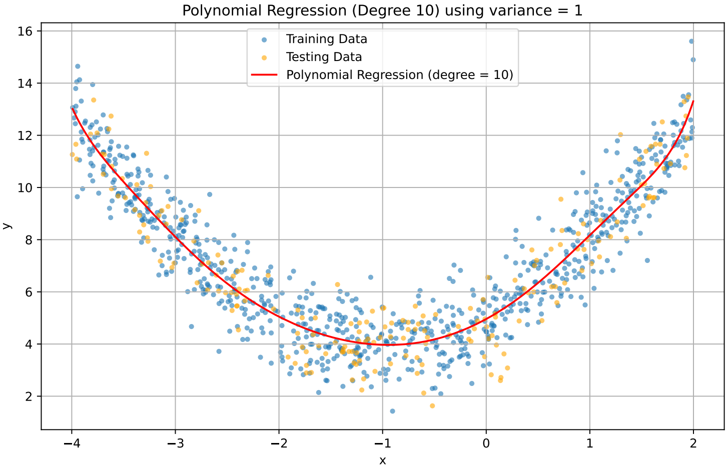
Implementation of Polynomial Regression in Scikit-Learn



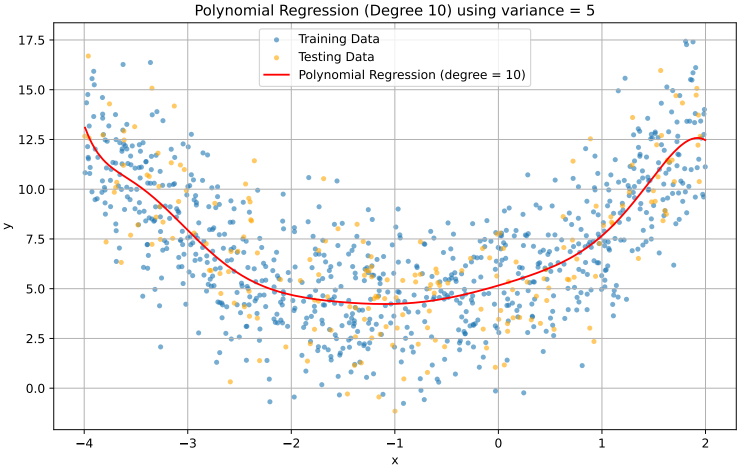
Implementation of Polynomial Regression in Scikit-Learn



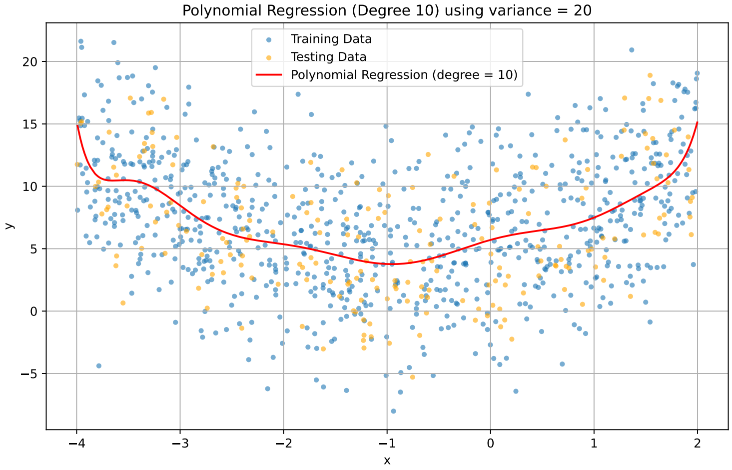
Implementation of Polynomial Regression in Scikit-Learn



Implementation of Polynomial Regression in Scikit-Learn



Implementation of Polynomial Regression in Scikit-Learn



Implementation of optimizing algorithms in different ML frameworks

1. **Scikit – Learn** :

   Stochastic gradient descent can be implemented using 'SGDRegressor'.

   ```
   sklearn.linear_model.SGDRegressor(loss='squared_loss',
   penalty='l2', alpha=0.0001, l1_ratio=0.15,
   fit_intercept=True, n_iter=5, shuffle=False, verbose=0,
   epsilon=0.1, random_state=None, learning_rate='invscaling',
   eta0=0.01, power_t=0.25, warm_start=False)
   ```

   The details of the arguments can be found here.

Implementation of optimizing algorithms in different ML frameworks

1. **Scikit – Learn** :

   The mini-batch gradient descent algorithm can be implemented via the feature called 'partial_fit' method.

   Unlike traditional estimators in Scikit-Learn that require the entire dataset to be provided at once for training (via the 'fit()' method), the 'partial_fit()' method allows for incremental learning.

   Below includes an example implementation of mini-batch gradient descent via Scikit-Learn.

```
import numpy as np
from sklearn.linear model import SGDRegressor

# Number of training points (for a large dataset)
num_samples = 100000
num_features = 10

# Function to generate data
```

Implementation of optimizing algorithms in different ML frameworks

```
# Function to generate data
def generate_data(num_samples, num_features):
X = np.random.rand(num_samples, num_features)
y = np.random.rand(num_samples)
return X, y

# Function to create mini batches and iterate
def iter_minibatches(X, y, chunksize):
start = 0
while start < 0 num_samples:
end = min(start + chuncksize, num_samples)
X_chunk, y_chunk = X[start:end], y[start:end]
yield X_chunk, y_chunk
start += chunksize

# training loop
def train_model(num_samples, num_features)
```

Implementation of optimizing algorithms in different ML frameworks

```
# training loop
def train_model(num_samples, num_features)

X, y  = generate_data(num_samples, num_features)

batch_iterations = iter_minibatches(X, y, chunksize = 32)
model = SGDRegressor()

for X_chunk, y_chunk in batch_iterations:
model.partial_fit(X_chunk, y_chunk)

return model

trained_model = train_model(num_samples, num_features)
```

Implementation of optimizing algorithms in different ML frameworks

2. **Keras** :

   In Keras there are many optimizers available. The details of the optimizers available in Keras can be found here. They can be specified in 'model.compile()' or 'model.fit()' methods.

   compile method

```
Model.compile( optimizer="rmsprop", loss=None,
loss_weights=None, metrics=None, weighted_metrics=None,
run_eagerly=False, steps_per_execution=1, jit_compile="auto",
auto_scale_loss=True, )
```

   The details of the arguments can be found here.

Implementation of optimizing algorithms in different ML frameworks

2. **Keras** :
   fit method

```
Model.fit(x=None, y=None, batch_size=None, epochs=1,
verbose="auto", callbacks=None, validation_split=0.0,
validation_data=None, shuffle=True, class_weight=None,
sample_weight=None, initial_epoch=0, steps_per_epoch=None,
validation_steps=None, validation_batch_size=None,
validation_freq=1,)
```

   The details of the arguments can be found here.

   **Note: The most direct way to implement Mini-Batch Gradient Descent in Keras is by specifying the batch_size parameter in the model.fit() method. This parameter determines the number of samples per gradient update. For instance, batch_size=32 will update model weights after every 32 samples.

Implementation of optimizing algorithms in different ML frameworks

3. **PyTorch** :

   In pytorch ‘torch.optm’ is a package implementing various optimization algorithms.

   To construct an Optimizer you have to give it an iterable containing the parameters (all should be Variable s) to optimize. Then, you can specify optimizer-specific options such as the learning rate, weight decay, etc.

   Example:

```
optimizer = optim.SGD(model.parameters(), lr=0.01,
 momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

   The basics of PyTorch implementations can be found here.

Implementation of optimizing algorithms in different ML frameworks

3. **PyTorch** :

   PyTorch provides tools like 'DataLoader' for easy implementation of Mini-Batch Gradient Descent. DataLoader handles data loading and preprocessing, streamlining the training process.

   'DataLoader' in PyTorch is a powerful utility that automates the process of dividing the data set into batches. It ensures that each mini-batch is correctly fed into the model during the training phase, optimizing the learning process.

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64,
shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64,
shuffle=True)
```