Assignment V: Non-Linear Probabilistic Regression Models

Introduction to Machine Learning Lab (190.013), SS2023
Björn Ellensohn[1]

[1]*m01435615, bjoern.ellensohn@stud.unileoben.ac.at*, *Montanuniversität Leoben, Austria*

May 25, 2023

This document guides through the process of solving Assignment 5.

1 Introduction

The 5^{th} assignment's task offers us with a more advanced look at machine learning techniques. This time it was allowed to use scikit-learn which makes life much easier for a data scientist. So it is even possible to directly import datasets from scikit library. In this assignment, we are evaluating the "iris" dataset from scikit-learn and the "concrete" dataset from UCI. In the end, the algorithms that had to be integrated where:

- Perceptron Algorithm
- Polynomial Regression
- Radial Basis Functions Regression
- Multilayer Perceptron Algorithm (Bonus)

I will walk through the code in 4 Sections.

2 Part I - Perceptron Algorithm

2.1 First Things First

As a guide through the assignment, an unsolved Jupyter Notebook was given, so we had an outline on how to solve this task. Before the actual code starts, dependencies had to be installed via `pip install`. Also, the iris dataset is loaded from scikit-learn via `load_iris()`.

2.2 Preparing the Data

Data preprocessing is an essential part of machine learning tasks. The dataset has to be examined for outliers and missing values. Outliers can easily be found using `DataFrame.boxplot()` and doing a 1.5 x IQR analysis. Outliers and missing values are then mitigated by selecting an approach which must be adapted to the type of data given. One simple method is replacing the outlier or missing data with the mean value of the column. After that is done, the dataset must be split into features and targets, which are in turn separated into training and testing sets. So in the end we will end up with four datasets:

- X_train
- X_test
- y_train
- y_test

Where the *class_train sets are used to evaluate the performance of the regression methods later.

2.3 Defining the Perceptron Algorithm

The Perceptron Algorithm, is the most basic single-layered neural network used for binary classification. This algorithm was inspired by the basic processing units in the brain, called neurons, and how they process signals. A visualization as of how the model works is seen in Figure 1.

The formula for the Perceptron update rule is displayed in Figure 2.

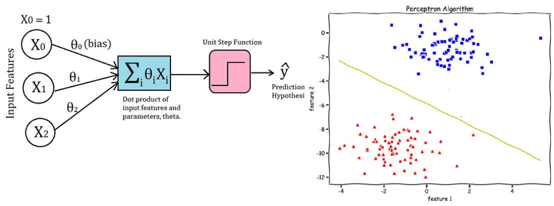

Figure 1: *Perceptron Algorithm Description.*

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}.$$

Figure 2: *The Formula for Perceptron Algorithm.*

For the implementation in Python a class `MultiClassPerceptron` is created which takes as inputs the input dimensions of provided data, the wanted output dimensions, the learning rate and the number of epochs. The forward, backward and predict functions had to be implemented.

In this implementation, the forward() method calculates the weighted sum of inputs and biases, applies the softmax function to obtain class probabilities, and returns the probabilities. The backward() method calculates the gradients of the weights and biases using the predicted probabilities and the true labels, and then updates the parameters using gradient descent. The predict() method uses the forward() method to get the class probabilities and selects the class with the highest probability as the predicted class for each example.

2.3.1 Forward Function

```
def forward(self, X):
    weighted_sum = np.dot(X, self.W) + self.b
    probabilities = np.exp(weighted_sum) /
    ↪  np.sum(np.exp(weighted_sum), axis=1,
    ↪  keepdims=True)
    return probabilities
```

2.3.2 Backward Function

```
def backward(self, X, y):
    m = X.shape[0]

    probabilities = self.forward(X)
    # Convert y to one-hot encoded form
    y_one_hot = np.eye(self.W.shape[1])[y]

    dW = (1 / m) * np.dot(X.T, (probabilities
    ↪  - y_one_hot))
    db = (1 / m) * np.sum(probabilities -
    ↪  y_one_hot, axis=0)

    self.W -= self.lr * dW
    self.b -= self.lr * db
```

2.3.3 Predict Function

```
def predict(self, X):
    probabilities = self.forward(X)
    predictions = np.argmax(probabilities,
    ↪  axis=1)
    return predictions
```

2.4 Evaluation

Having the code done, the model is then be trained and evaluated. For performance analysis, accuracy_score method from the scikit-learn metrics package is used. Having tuned the lr parameter to 0.0005 the Perceptron classification train accuracy is 0.6666666666666666 and the Perceptron classification accuracy is 0.7. However, when re-running the training, the results change so I assume there must be a problem somewhere.

3 Part II - Polynomial Regression

3.1 Introduction

If your data points are not connectable by a straight line, Polynomial Regression might help you out. Figure 3 gives a nice example how this looks like.

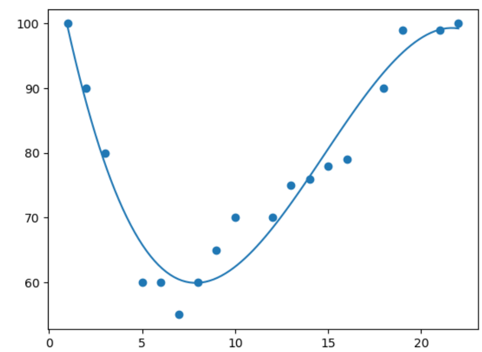

Figure 3: *Exemplary Display of Polynomial Regression.*

3.2 Preparing the Data

This time, the "concrete" dataset from UCI shall be processed and evaluated. Since the data is provided in an xls file, it must first be imported into a Pandas dataframe. After that, the usual feature and target selection is taking place. The concrete's compressive strength is selected as target, as this is connected to all the other parameters of this dataset. So the other columns are threated as features. In the code, this is achieved by using the `DataFrame.drop()` method and slicing.

3.3 Implementing the Polynomial Regression

The data preparation part once again leads to four subsets:

- X_train
- X_test
- y_train
- y_test

Next, the features had to be transformed into polynomials. This is done using a hand-made function called `polynomial_features()`, which takes as inputs the desired polynomial's degree and the subset to process.

The function iterates over the combinations with replacement of feature indices up to the specified degree using `combinations_with_replacement()`. For each combination, it creates a new array of polynomial features by multiplying the corresponding columns of X. The resulting polynomial features are appended to the `polynomial_X` list.

Finally, the list of polynomial features is stacked horizontally using `np.column_stack()` to form the final `polynomial_X` array, which is then returned. The code for that follows.

3.3.1 Polynomial Features Function

```python
def polynomial_features(X, degree):
n_samples, n_features = X.shape
polynomial_X = np.ones((n_samples, 1))

for d in range(1, degree + 1):
    combinations = combinations_with_replacement(range(n_features), d)
    for comb in combinations:
        poly_features = np.prod(X[:, comb], axis=1, keepdims=True)
        polynomial_X = np.hstack((polynomial_X, poly_features))

return polynomial_X
```

3.3.2 Training the Model

After that, the LinearRegression class from sklearn.linear_model is used to make life easier but we will extend it by using our `polynomial_features()`. Now we are almost ready to train our model, only some minor steps need to be taken. Let's see the code:

```python
# Train the model
lr_poly_custom = LinearRegression()
lr = LinearRegression()
# fit the model
lr_poly_custom.fit(X_train_poly_custom, y_train)
lr.fit(X_train, y_train)
# predict values from the polynomial transformed features
predictions_poly_custom_train = lr_poly_custom.predict(X_train_poly_custom)
predictions_poly_custom = lr_poly_custom.predict(X_test_poly_custom)
# predict values from the original features
predictions_train = lr.predict(X_train)
predictions = lr.predict(X_test)
```

3.3.3 Evaluation

To better get a feel for the Polynomial Regression's performance, it is also compared to a standard linear regression model on the same dataset. The results are displayed in Figure 4.

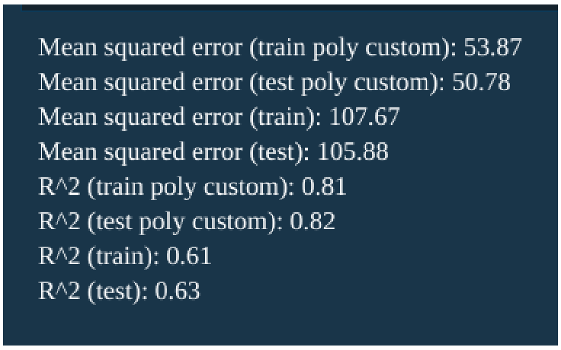
Mean squared error (train poly custom): 53.87
Mean squared error (test poly custom): 50.78
Mean squared error (train): 107.67
Mean squared error (test): 105.88
R^2 (train poly custom): 0.81
R^2 (test poly custom): 0.82
R^2 (train): 0.61
R^2 (test): 0.63

Figure 4: *Evaluation of Polynomial Regression compared to Linear Regression.*

Comparing the mean squared error values, it is easy to say that Polynomial Regression performance is way better in this case.

4 Part III - Radial Basis Functions Regression

Another way to implement non-linear Regression is using Radial Basis Functions (RBFs).

4.1 Introduction

For the vast variety of datasets out there it is always good to have some specialized regression models at hand. The RBF Regression follows formula (1).

$$h(x) = \sum_{n=1}^{N} w_n \times exp(-\gamma \|x - x_n\|^2) \qquad (1)$$

A visualization of RBFs can be seen in figure 5.

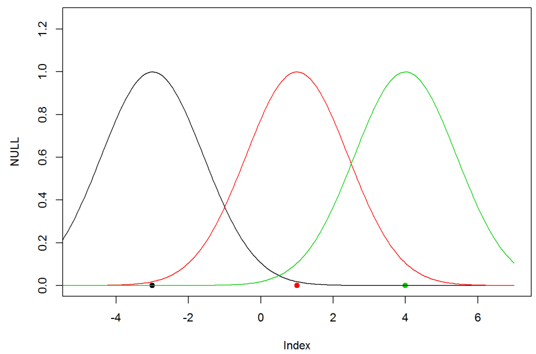

Figure 5: *Evaluation of Polynomial Regression compared to Linear Regression.*

4.2 Walktrough RBFs

For this part of the assignment, the California Housing Prices dataset is used. It is included into scikit-learn and can be imported by calling the `fetch_california_housing()` method.

Again, `train_test_split()` is used to produce some fine subsets for the following data-science-fun. As a preparation step the data standardized too. The `StandardScaler` class is best-suited for this kind of work:

```
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.fit_transform(X_test)
```

The most important part was implementing the `rbf_kernel()` function so i print it right here:

```
def rbf_kernel(X, centers, gamma):
    n_samples = X.shape[0]
    n_centers = centers.shape[0]
    K = np.zeros((n_samples, n_centers))

    for i in range(n_samples):
        for j in range(n_centers):
            diff = X[i] - centers[j]
            K[i, j] = np.exp(-gamma *
            ↪  np.dot(diff, diff))

    return K
```

The remaining steps are as following. As always, the full code is attached in the appendix, so a short summary will do for now.

1. Choose the number of centroids and the RBF kernel width
2. Randomly select the centroids from the training set
3. Compute the RBF features for the training and testing sets
4. Fit a linear regression model on the original and RBF-transformed data

After these steps, the evaluation could take place.

4.3 Evaluation

The RBF Regression is compared against the normal Linear Regression. To my surprise, the results only show a minor difference between those two methods. This is pictured in figure 6.

```
Linear regression on original data:
MSE: 0.5188091411147203
R^2: 0.6005669603670689

Linear regression on RBF-transformed data:
MSE: 0.5439916082749939
R^2: 0.5811788875554136
```

Figure 6: *Evaluation of RBF Regression compared to Linear Regression.*

5 Part IV - Multilayer Perceptron Algorithm (Bonus)

The bonus part was skipped so there is not much to say here.

APPENDIX

The complete code is following.

Code for Part I

```
# load the iris dataset
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
import numpy as np

iris = load_iris()
X = iris.data
y = iris.target

# Preprocess the data
from sklearn.model_selection import train_test_split

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.8,
↪  random_state=None, shuffle=True, stratify=None)

import numpy as np

# Define the perceptron algorithm
class MultiClassPerceptron:
    def __init__(self, input_dim, output_dim, lr=0.01, epochs=1000):
        self.W = np.random.randn(input_dim, output_dim)
        self.b = np.zeros((1, output_dim))
        self.lr = lr
        self.epochs = epochs

    def forward(self, X):
        weighted_sum = np.dot(X, self.W) + self.b
        probabilities = np.exp(weighted_sum) / np.sum(np.exp(weighted_sum), axis=1, keepdims=True)
        return probabilities

    def backward(self, X, y):
        m = X.shape[0]

        probabilities = self.forward(X)
        # Convert y to one-hot encoded form
        y_one_hot = np.eye(self.W.shape[1])[y]

        dW = (1 / m) * np.dot(X.T, (probabilities - y_one_hot))
        db = (1 / m) * np.sum(probabilities - y_one_hot, axis=0)

        self.W -= self.lr * dW
        self.b -= self.lr * db

    def fit(self, X, y):
        for epoch in range(self.epochs):
            self.forward(X)
            self.backward(X, y)

    def predict(self, X):
        probabilities = self.forward(X)
        predictions = np.argmax(probabilities, axis=1)
        return predictions

# Train the model
p = MultiClassPerceptron(input_dim=X_train.shape[1], output_dim=3, lr=0.0005, epochs=1000)
p.fit(X_train, y_train)
```

```
predictions_train = p.predict(X_train)
predictions = p.predict(X_test)

# evaluate train accuracy
print("Perceptron classification train accuracy", accuracy_score(y_train, predictions_train))
print("Perceptron classification accuracy", accuracy_score(y_test, predictions))
```

Code for Part II

```
import numpy as np
from itertools import combinations_with_replacement

# Implement the polynomial_features() function
def polynomial_features(X, degree):
    n_samples, n_features = X.shape
    polynomial_X = np.ones((n_samples, 1))

    for d in range(1, degree + 1):
        combinations = combinations_with_replacement(range(n_features), d)
        for comb in combinations:
            poly_features = np.prod(X[:, comb], axis=1, keepdims=True)
            polynomial_X = np.hstack((polynomial_X, poly_features))

    return polynomial_X

# Non-linear feature transformation
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# load the concrete compressive strength dataset
df = pd.read_excel('Concrete_Data.xls')

# Selecting the Concrete compressive strength as targets, rest is features
X = df.drop('Concrete compressive strength(MPa, megapascals) ', axis=1)
y = df['Concrete compressive strength(MPa, megapascals) ']


# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.8,
↪  random_state=None, shuffle=True, stratify=None)

# transform the features into second degree polynomial features
X_train_poly_custom = polynomial_features(X_train.values, degree=2)
X_test_poly_custom = polynomial_features(X_test.values, degree=2)

# Train the model
lr_poly_custom = LinearRegression()
lr = LinearRegression()
# fit the model
lr_poly_custom.fit(X_train_poly_custom, y_train)
lr.fit(X_train, y_train)
# predict values from the polynomial transformed features
predictions_poly_custom_train = lr_poly_custom.predict(X_train_poly_custom)
predictions_poly_custom = lr_poly_custom.predict(X_test_poly_custom)
# predict values from the original features
predictions_train = lr.predict(X_train)
predictions = lr.predict(X_test)

# mean squared error
```

```
print("Mean squared error (train poly custom): {:.2f}".format(mean_squared_error(y_train,
↪   predictions_poly_custom_train)))
print("Mean squared error (test poly custom): {:.2f}".format(mean_squared_error(y_test,
↪   predictions_poly_custom)))
print("Mean squared error (train): {:.2f}".format(mean_squared_error(y_train, predictions_train)))
print("Mean squared error (test): {:.2f}".format(mean_squared_error(y_test, predictions)))

# coefficient of determination (R^2)
print("R^2 (train poly custom): {:.2f}".format(r2_score(y_train, predictions_poly_custom_train)))
print("R^2 (test poly custom): {:.2f}".format(r2_score(y_test, predictions_poly_custom)))
print("R^2 (train): {:.2f}".format(r2_score(y_train, predictions_train)))
print("R^2 (test): {:.2f}".format(r2_score(y_test, predictions)))

```

Code for Part III

```
import numpy as np

# Implement the rbf_kernel() function
def rbf_kernel(X, centers, gamma):
    n_samples = X.shape[0]
    n_centers = centers.shape[0]
    K = np.zeros((n_samples, n_centers))

    for i in range(n_samples):
        for j in range(n_centers):
            diff = X[i] - centers[j]
            K[i, j] = np.exp(-gamma * np.dot(diff, diff))

    return K

from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Load the California Housing Prices dataset
data = fetch_california_housing()
X = data['data']
y = data['target']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.8,
↪   random_state=None, shuffle=True, stratify=None)

# Standardize the data
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.fit_transform(X_test)

# Choose the number of centroids and the RBF kernel width
num_centroids = 100
gamma = 0.1

# Randomly select the centroids from the training set
np.random.seed(42)
idx = np.random.choice(X_train_std.shape[0], num_centroids, replace=False)
centroids = X_train_std[idx]

# Compute the RBF features for the training and testing sets
rbf_train = rbf_kernel(X_train_std, centroids, gamma)
```

```
rbf_test = rbf_kernel(X_test_std, centroids, gamma)

# Fit a linear regression model on the original and RBF-transformed data
linreg_orig = LinearRegression().fit(X_train_std, y_train)
linreg_rbf = LinearRegression().fit(rbf_train, y_train)

# Evaluate the models on the testing set
y_pred_orig = linreg_orig.predict(X_test_std)
mse_orig = mean_squared_error(y_test, y_pred_orig)
r2_orig = r2_score(y_test, y_pred_orig)

y_pred_rbf = linreg_rbf.predict(rbf_test)
mse_rbf = mean_squared_error(y_test, y_pred_rbf)
r2_rbf = r2_score(y_test, y_pred_rbf)

# Print the results
print("Linear regression on original data:")
print("MSE:", mse_orig)
print("R^2:", r2_orig)

print("\nLinear regression on RBF-transformed data:")
print("MSE:", mse_rbf)
print("R^2:", r2_rbf)
```