CME 466: Design of a Digital System

Winter 2024

Machine Learning – Linear Regression [part 1]

KHAN A. WAHID, PhD, PEng, SMIEEE

Professor
Electrical and Computer Engineering
University of Saskatchewan
Email: khan.wahid@usask.ca

1. What is Linear Regression?

- The target (or predicted) value is expected to be a linear combination of the features (or coefficients). In mathematical notation, if y is the predicted value,

$$y(w, x) = w_0 + w_1x_1 + \cdots + w_nx_n$$

  Here, w_0 is the intercept and $w_1, w_2, \ldots, w_n$ are the coefficients.

- In scikit-learn, Linear Regression uses an **Ordinary Least Squares**[1] model that fits a linear model with coefficients $w = (w_1, w_2, \ldots, w_n)$ to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation. Therefore, the relation becomes (for one feature):

$$y(w, x) = w_0 + w_1x_1 + \varepsilon$$

  Where, ε is the residual error, w_0 is the intercept and w_1 is the slope of the line.

- The loss function we want to minimize is:

$$\sum_{i=1}^{n} (y_i - (w_0 + w_ix_i))^2$$

- More information about the parameters and attributes can be found below:

  https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression

[1] https://www.sfu.ca/~dsignori/buec333/lecture%208.pdf

1.1 scikit-learn Datasets (using Diabetes dataset)

In our class, we will use some standard datasets from scikit-learn that do not require to be downloaded from any external website. The datasets are here:

https://scikit-learn.org/stable/datasets.html

- The first one is the "Toy" datasets[2]. Let us look at the **Diabetes dataset** in this example.
- It is a database of 442 diabetes patients with 10 attributes (age, sex, bmi, bp, etc.) in the first 10 columns. The 11th column is a quantitative measure of disease progression one year after baseline. The original database is here[3].
- Note that each of these 10 feature variables have been mean centered and scaled by the standard deviation times the square root of 442 (i.e. the sum of squares of each column totals 1).
- Launch Jupyter Notebook in the virtualenv and run the following codes to observe the differences between the actual dataset and the scaled (or normalized) dataset.

```
# load and save diabetes data set
from sklearn.datasets import load_diabetes
diabDS = load_diabetes()
type(diabDS)
# see how the data sets are internally stored and constructed by scikit-learn
print(diabDS.data.shape)
# retrieve the descriptive feature names
print(diabDS.feature_names)
```

[2] https://scikit-learn.org/stable/datasets/toy_dataset.html
[3] https://www4.stat.ncsu.edu/~boos/var.select/diabetes.tab.txt

- The data is loaded as a Bunch class[4], which is a dictionary that supports attribute-style access. Now, we need to load the data (i.e., features) and targets into two separate variables/arrays.

```
# If True, returns (data, target) instead of a Bunch object.
diabDS_X, diabDS_y = load_diabetes(return_X_y=True)
print(diabDS_X)
print(diabDS_X.data.shape)
```

- We can see some data samples and target values.

```
diabDS_X[0:5]
```

```
diabDS_y[0:5]
```

- Carefully check the data array. Why are the values different than the actual values (compare with the original database)?

- We can also use panda dataframe to display the dataset in a more organized way. However, let us not use panda for now.

```
import pandas as pd
df = pd.DataFrame(diabDS.data, columns = diabDS.feature_names)
df['target'] = diabDS.target
df
```

[4] https://pypi.org/project/bunch/

1.2 Example of Linear Regression using Diabetes dataset

- Before we move further, we need to import several libraries as given below:

```
import numpy as np
import matplotlib.pyplot as plt
# import linear regression class from scikit-learn
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()

# load evaluation parameters[5]
from sklearn.metrics import mean_squared_error, r2_score
mse=mean_squared_error
r2=r2_score

# test mse and r2
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]

mse(y_true, y_pred)

r2(y_true, y_pred)
```

- Here, R^2 is the coefficient of determination or regression score function. It is the proportion of the variation in the dependent variable that is predictable from the independent variable(s). The best possible score is 1.0.

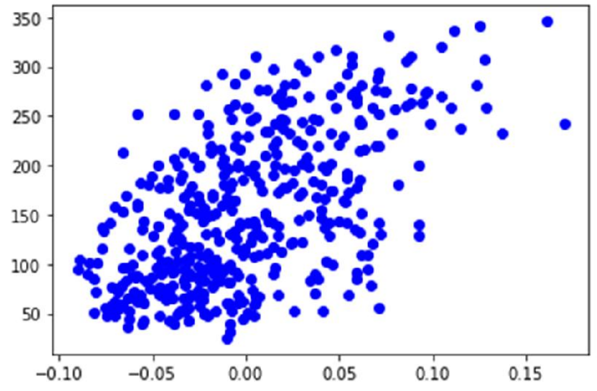
$$R^2 = 1 - \frac{SS_{residual}}{SS_{total}}$$

[5] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html

- Coming back to our dataset, the data loaded into diabDS is a 442 x 10 matrix. Here, there are 10 features in the dataset (age, sex, …, etc.). Firstly, we will choose only one feature out of the 10 to train the model. Do the following:

```
# take only one feature to train (0=age, 1=sex, 2-bmi,...)
diabDS_x = diabDS_X[:, np.newaxis, 2]
print(diabDS_x.data.shape)
print(diabDS_x)
```

- **Exploratory data analysis:** This is an important step in the design. Before we go further and train our model, we should observe the data points. This will give us an idea about how the data points are spread out. This is known as exploratory data analysis. We can then decide intuitively whether to keep the feature and/or which model or estimator to use.



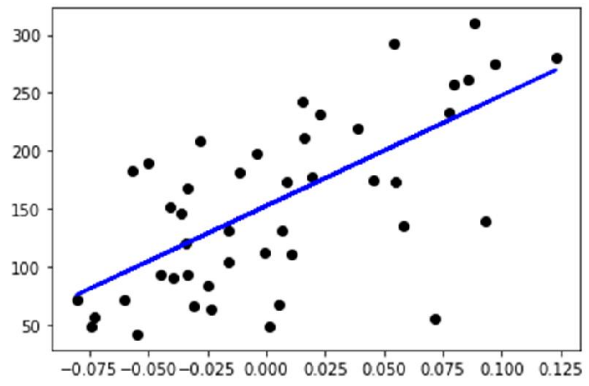
```
# see all data and target
plt.scatter(diabDS_x, diabDS_y, color="blue")
```

- The next stage is of the design is to split the entire database into training and test sets. Since we have only one dataset, we have to split it. Reusing the same data points for testing that were used during training will not truly evaluate the effectiveness of your model.

- The more the data points are available for training, the better it is. However, we **also** need to keep a good amount data point for testing to properly evaluate the model. Therefore, a common practice is to use 75% to 80% of the data points for training. The rest is kept for testing.

- In our example, we only keep 45 samples for testing. The rest is used for training purpose.

```
# use last 45 data samples as test, rest as train
diabDS_xtrain = diabDS_x[:-45]
diabDS_xtest = diabDS_x[-45:]

# just to check
print(diabDS_xtest)

# use last 45 target samples as test, rest as train
diabDS_ytrain = diabDS_y[:-45]
diabDS_ytest = diabDS_y[-45:]

# just to check
print(diabDS_ytest)

# to make sure, see training data and target
plt.scatter(diabDS_xtrain, diabDS_ytrain, color="black")

# train the classifier with the train dataset
lin_reg.fit(diabDS_xtrain,diabDS_ytrain)

# now time to make predictions using the model
diabDS_ypred = lin_reg.predict(diabDS_xtest)
print(diabDS_ypred)
```



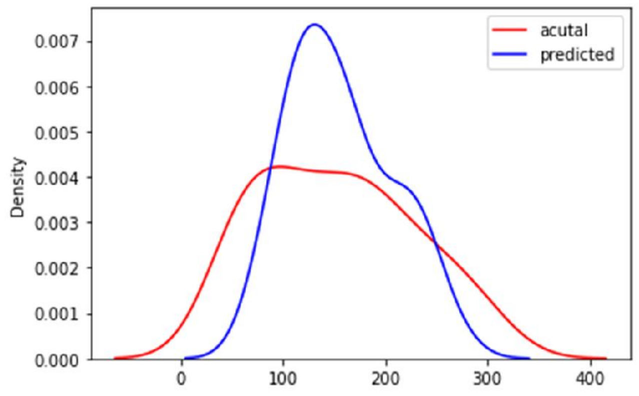
```
# plot test data points, then plot the decision line on top
plt.scatter(diabDS_xtest, diabDS_ytest, color="black")
plt.plot(diabDS_xtest, diabDS_ypred, color='blue', linewidth=2)


# see evaluation parameters
```

```
print("Coefficients: \n", lin_reg.coef_)
print("Intercept: \n", lin_reg.intercept_)
print("Mean squared error: %.2f" % mse(diabDS_ytest, diabDS_ypred))
print("Coefficient of determination: %.2f" % r2(diabDS_ytest, diabDS_ypred))
```

```
    Coefficients:
     [953.02484976]
    Mean squared error: 3203.71
    Coefficient of determination: 0.42
```

- We can use a kernel density estimate (KDE)[6] plot to see the distribution of both actual and predicted values (same as histogram). KDE represents the data using a continuous probability density curve in one or more dimensions.

```
# see kernel density estimate (KDE) plot
import seaborn as sns
sns.kdeplot(diabDS_ytest, color='red',
label='acutal')
sns.kdeplot(diabDS_ypred, color='blue',
label='predicted')
plt.legend()
```



[6] https://seaborn.pydata.org/generated/seaborn.kdeplot.html

- **Questions:**
    - Are we satisfied with a low coefficient of determination[7] ?
    - Can we use a different feature for training and see if performance improves or not
    - Is Linear regression a good model in this case (Diabetes dataset)?

1.5 Other Linear Regression Models in scikit-learn

- There are several other models that are supported in scikit-learn[8]. Two important ones are **Ridge regression** and **Lasso regression**. We can change our code and use these models to check the performance. The difference between these methods and ordinary linear regression (OLS) is that they use a regularization term (or penalty) along with the linear least squares function in their loss function.

- This penalty term modulates the slope of the fitted line. It adds as a tradeoff between fitting the training data well while keeping the parameters small.

- **Ridge regression** uses L2 regularization. The loss function for Ridge regression is:

$$\sum_{i=1}^{n}(y_i - (w_0 + w_i x_i))^2 + \sigma \sum_{j=1}^{p} w_j^2$$

[7] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html
[8] https://scikit-learn.org/stable/modules/linear_model.html

- **Lasso regression** uses L1 regularization. The loss function for Lasso regression is:

$$\sum_{i=1}^{n}(y_i - (w_0 + w_i x_i))^2 + \sigma \sum_{j=1}^{p}|w_j|$$

- Parameter σ (alpha) is used to control the weight of the regularization. By making alpha = 0, both Ridge and Lasso become OLS regression.

```
# other regression models
from sklearn import linear_model
lin_reg = linear_model.Ridge(alpha=0.1)
# lin_reg = linear_model.Lasso(alpha=0.1)
print(lin_reg)
```