KU Leuven

Machine Learning: A brief Overview

Tom De Groote

January 19, 2017

Contents

Chapter 1

Introduction

Chapter 2

Regression

Regression is a form of supervised machine learning with as goal to take continuous data and find the equation that best fits the data. This way you'll be able to forecast a specific value.

2.1 Linear Regression

The best fit function searched is just a linear line. See figure 2.1 for an example. Since linear regression is the basis of almost all machine learning algorithms (it is also used in Neural Networks for example), we will elaborate a bit more on how it actually works.
As we know a first order line can be simply represented as $y = mx + b$. We know x since it are our labels and when training we also know y since it are our features (which we know, since it's supervised learning). So the goal of linear regression is to calculate m and b, calculating m is achieved with the following formula: $m = \dfrac{\overline{x} \cdot \overline{y} - \overline{xy}}{(\overline{x})^2 - \overline{x^2}}$ where the bar over the letters signifies a mean or average. To calculate b the following formaly can be used: $b = \overline{y} - m\overline{x}$. When you use these formulas to calculate the regression line you are actually minimising the sqaured error between the regression line's y values and the data's y values. To know how well the regression line predicts the data's y values you can check the outcome of the *r squared method*, see chapter 5.

2.1.1 Code examples

Two code approaches have been made. The first approach uses the *sklearn* kit for doing linear regression as well as experimenting with some support vector machines, the approach can be found in appendix A. The second approach shows a more basic linear regression which illustrates it's fundamentals. Since linear regression is the basis of a lot of machine learning algorithms, this code can help you understand the basic building blocks of all these machine learning alorithms. It also shows how the *r squared method* works. This code can be found in appendix B.

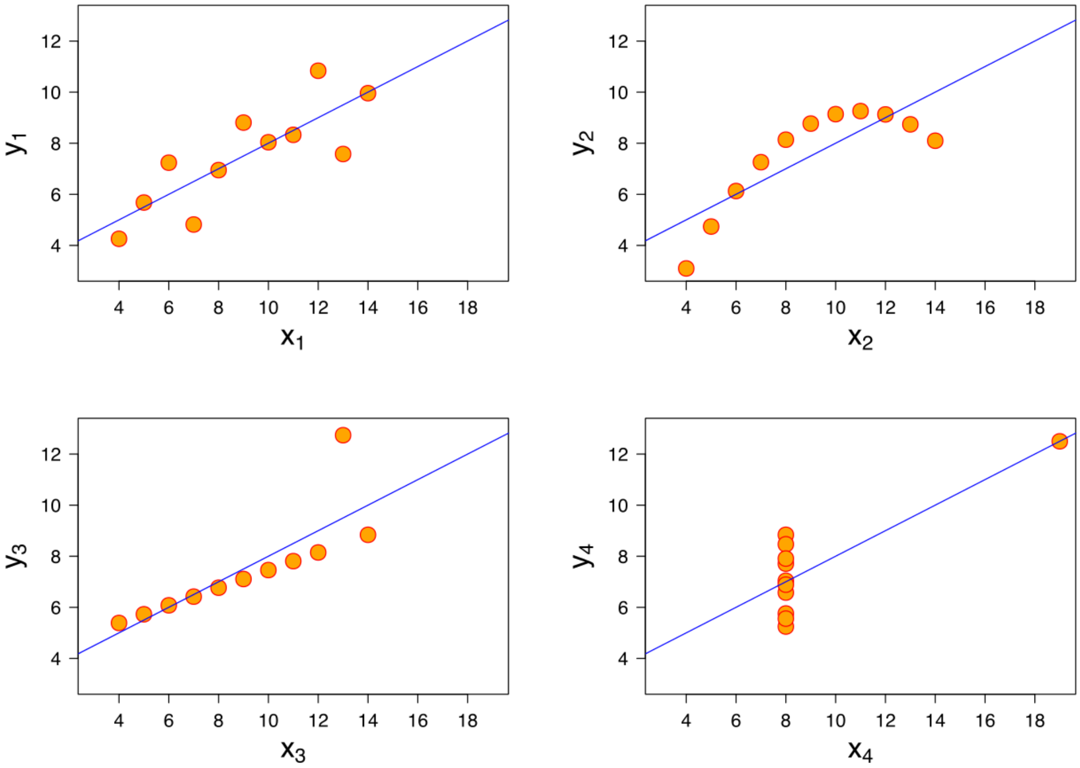

Figure 2.1: Four examples of the function found by linear regression based on the given data points.

Chapter 3

Classification

Classification is a form of supervised machine learning (in contrary to Clustering, see chapter 4. It takes examples which we have identified whith classes and tries to learn a model that will predict the class of unknown examples. An example of use is to classify tumors as benign or malignant. We feed the classifier the features, such as size and shape, of known results. After the learning phase we can then use this classifier to predict if a given tumor is benign or not.

3.1 K Nearest Neighbors

K nearest neighbors is a simple but effective classification algorithm. The algorithm works by finding the k neirest neighbors of a given data point and chosing a class based on the labels of these k nearest neighbors. Basically using the majority vote of these neighbors to choose the data point's class. It is also possible to assign weight to the vote of the neighbors based on their distance.
A known pitfall for the K Nearest Neighbor algorithm is that it needs to compare the data in question to all of the points from the dataset before we can know what the closest three points are. Therefor accuracy is easy to accomplish, but being fast is hard. Another way is to compare your data only to data within a certain radius. Other pitfalls include: problems with outliers and bad data.
The confidence of this algorithm can be measured in two ways:

- Correct versus incorrect
- Check the average vote confidence

3.1.1 Code examples

Two code approaches have been made. The first approach uses the *sklearn* kit, the approach can be found in appendix C. The second approach shows a more basic KNN which illustrates it's fundamentals. This code can help you understand the basic building blocks of the algorithm and let you see where it's pitfalls are. The code can be found in appendix D.

3.2 Support Vector Machines

A SVM is a bineairy classifier. It tries to classify the examples by seperating them lineairly. This is done by maximising the distance between the seperating plane and the closest example of both classes, as shown in figure 3.1

3.2.1 Support Vector Machine Regression

It is possible to use SVMs to learn linear regression lines, as shown in our code example B, however we will not discuss in depth how this works.

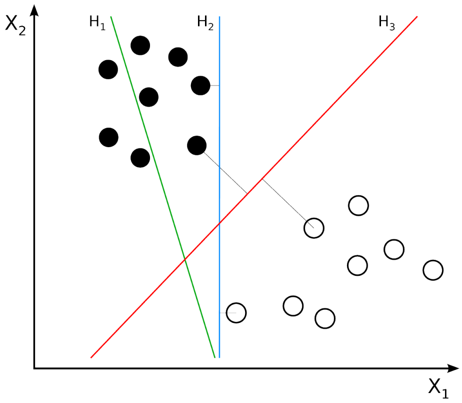

Figure 3.1: Shows three seperation planes, H1 is not a good seperating plane, H2 and H3 are acceptable.

Chapter 4

Clustering

Chapter 5

General Terms

Confidence Score A score that tells you how accurate and reliable a model is performing based on the test data.

Cross Validation Is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set. It splits the data set in test and training data.

Eucledian Distance A way to calculate the distance on a plane between points. It uses the following formula: $\sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$. Measures the length of a line segment between points.

Eucledian Norm Measures the magnitude of a vector, which is basically the length. The equation is also the same as with Eucledian Distance, the name just tells you what space you are using.

Features Descriptive attributes for the data.

Kernels Is a kind of transformation on your data. Grossly put it simplifies your data. More specifically kernel methods use kernel functions to operate in a high-dimensional, implicit feature space without ever computing the coordinates of the data in that space, but rather by simply computing the inner products between the images of all pairs of data in the feature space. This is computationally a lot better than using the raw data. For an example see figure 5.1.

Labels What you are trying to predict of forecast for the data.

Linear Algebra The objective of linear algebra is to calculate relationships of points in vector space.

Preprocessing Used to clean/scale the data before using machine learning techniques. Cleaning for example by replacing NaN data with -99 999, because it will be handled as an outlier, or by interpolating it. Scaling your features so they fall between -1 and 1 is genarally a good idea because it could make the processing faster and more accurate.

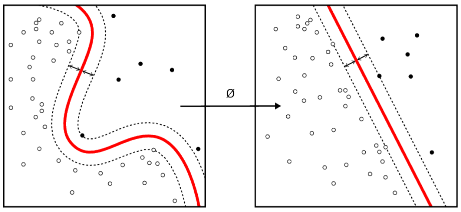

Figure 5.1: An example of simplifying the data by increasing it's dimension.

Machine Learning Classifier

Machine Learning Model

Supervised Learning For om machine learning where the scientist teaches the machine by showing it features and then showing it the correct answer (lable). Once the machine is taught, the scientis will usually test the machine on some unseen data, where the scientis still knows the correct answer, but the machine doesn't.

R squared method Also known as *coefficient of determination.* The squared error is the mean or sum of the distance between the solution values and the actual values. For example in Linear regression the error is the distance between the regresssion line's y values and the data's y values. The squared error is either a mean or sum of this. Squared error is used because on the one hand it normalises all errors to be positive and on the other hand it punishes outliers harder. Since the squared error is just a relative number to your dataset it has no real meaning, that's why we use the r squared method. This method uses the formula $r^2 = 1 - \frac{SE\hat{y}}{SE_{\overline{y}}}$ which is just one minus the division of the squared error of the regression line and the squared error of the mean y line. A number close to 1 means the classifier is performing well, a number close to 0 means it is performing bad. It is a good measure when trying to predict an exact future value, however if you just want to predict a general tendense it is not the best measure.

Threading Some machine learning algorithms can be split into multiple threads, this is often indicated by the *n_ jobs* parameter in python. Others don't have this luxurary and are known as running linear.

Types of Data With machine learning we can see our data in several groups. It is important that these groups do not overlap, since otherwise a bad representation of results could be shown.

- **Training data** is the data used to train your machine learning model.
- **Testing data** is the data used to test your machine learning model.
- **Validation data** is the data used to validate your machine learning model.

Appendices

Appendix A

Regression

In this appendix you can find the code for a linear regression implementation using *sklearn*, as well as some examples of SVMs used for regression.

```
import pandas as pd
import quandl, math
import numpy as np
from sklearn import preprocessing, cross_validation, svm
from sklearn.linear_model import LinearRegression
import datetime
import matplotlib.pyplot as plt
from matplotlib import style
# Use Pickle to save any python object
import pickle

# set the api key for quandl
quandl.ApiConfig.api_key = "ENzts_Lf48qsmWQC_xJb"

# Retrieves data from quandli
df = quandl.get("WIKI/GOOGL")

# Only keep relevant adjusted columns
df = df[['Adj.␣Open', 'Adj.␣High', 'Adj.␣Low', 'Adj.␣Close', 'Adj.␣Volume']]

# Manipulate data so we can get useful information out of it
# First get the High Low Percentage
df['HL_PCT'] = (df['Adj.␣High'] - df['Adj.␣Low'])/df['Adj.␣Low']*100.0
# Next get the Daily Percentage
df['PCT_change'] = (df['Adj.␣Close'] - df['Adj.␣Open']) / df['Adj.␣Open'] * 100.0
# And change the data frame to represent these changes, throw away irrelevant data
df = df[['Adj.␣Close', 'HL_PCT', 'PCT_change', 'Adj.␣Volume']]

# Define the column we will try to forecast
forecast_col = 'Adj.␣Close'
# Replace the NaN values in the data with -99999
# Reason for this number is that most of the time it will be handled as an outlier.
df.fillna(value=-99999, inplace=True)
# How far do you want to forecast
forecast_out = int(math.ceil(0.01 * len(df)))

# All current columns are features, so we need to add a label column, shift is so the
# is the value of Adj. Close of the 1%th data point
df['label'] = df[forecast_col].shift(-forecast_out)

# sklearn needs numpy arrays for the machine learning part. But we did data manipulat
# Features are represented by X
X = np.array(df.drop(['label'], 1))

# Scaling the data
X = preprocessing.scale(X)
# Contains the most recent features, which we will predict against
X_lately = X[-forecast_out:]
# Only take X to the point we have known data labels
X = X[:-forecast_out]

# Drop all NaN created by the above actions
df.dropna(inplace=True)

# Labels are represented by y
y = np.array(df['label'])

# Splitting the data in test and train data
X_train, X_test, y_train, y_test = cross_validation.train_test_split(X, y, test_size=

# Use linear regression to define a classifier
# n_jobs identifies the number of threads that can be made, -1 identifies this to be
clf = LinearRegression(n_jobs=-1)

# Train the classifier
clf.fit(X_train, y_train)

# Calculate the confidence of our classifier
confidence = clf.score(X_test, y_test)

# Show our confidence score using linear regression
print("linear␣regression:", confidence)

# We will also experiment with some SVM's with different kernel functions
for k in ['linear', 'poly', 'rbf', 'sigmoid']:
    clf_extra = svm.SVR(kernel=k)
    clf_extra.fit(X_train, y_train)
    confidence = clf_extra.score(X_test, y_test)
    print(k, ":␣", confidence)

# We will move forward with the classifier clf from LinearRegression

# Calculate our forecast out
forecast_set = clf.predict(X_lately)
# Add a forecast column to dataframe
df['Forecast'] = np.nan

# Add the forecast data on the correct points
last_date = df.iloc[-1].name
last_unix = last_date.timestamp()
one_day = 86400
next_unix = last_unix + one_day
for i in forecast_set:
    # See what the next forecast date is
    next_date = datetime.datetime.fromtimestamp(next_unix)
    next_unix += one_day
    # Set all the columns to nans on forecast dates, except the forecast column, set
    df.loc[next_date] = [np.nan for _ in range(len(df.columns)-1)] + [i]
```

```
# Save our learned classifier using pickle
with open('LinearRegression/linearregression.pickle', 'wb') as f:
    pickle.dump(clf, f)
# To use the saved classifier just use the following commented line:
# pickle_in = open('LinearRegression/linearregression.pickle', 'rb')
# clf = pickle.load(pickle_in)

# Let's visualise
# Set the style of our graph
style.use('ggplot')
# Make the graph
df['Adj.␣Close'].plot()
df['Forecast'].plot()
plt.legend(loc=4)
plt.xlabel('Date')
plt.ylabel('Price')
plt.show()
```

Appendix B

Manual Regression

In this appendix you find a linear regression algorithm build from the ground up.

```
from statistics import mean
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
from DataGeneration import create_data_set

# Define the plotting style
style.use('ggplot')

# Define some starting points, we use the DataGeneration class to generate the data s
size = 40
variance = 10
xs, ys = create_data_set(size, variance, step=2, correlation='False')


# A function that given our x and y calculates the best fitting slope and the best fit
def best_fit_slope_and_intercept(xs, ys):
    m = (mean(xs)*mean(ys) - mean(xs*ys)) / (mean(xs)**2 - mean(xs*xs))
    b = mean(ys) - m*mean(xs)
    return m, b

# Get the best fitting slope and the
m, b = best_fit_slope_and_intercept(xs, ys)

# Show our best fit slope and y-intercept
print("m␣and␣b", m, b)

# Create the regression line
regression_line = [(m*x) + b for x in xs]

# Let's predict some points based on the regression line
# Our feature
predict_x = size + 5
# The predicted label
predict_y = (m * predict_x) + b
# Print the predicted label
print("Predicted␣y:", predict_y, "for␣x:", predict_x)


# Calculates the squared error of given original vector and the predicted vector
def squared_error(ys_orig, ys_line):
    return sum((ys_line-ys_orig)*(ys_line-ys_orig))
```

```
# Calculates the coefficient of determination given the original vector and the predi
def coefficient_of_determination(ys_orig, ys_line):
    # Create a line that is just a constant function of the average of the original y
    y_mean_line = [mean(ys_orig)] * len(ys_orig)
    # Calculate the top part of the r squared method equation
    squared_error_regr = squared_error(ys_orig, ys_line)
    # Calculate the bottom part of the r squared method equation
    squared_error_y_mean = squared_error(ys_orig, y_mean_line)
    # Calculate the full r squared method equation
    return 1 - (squared_error_regr/squared_error_y_mean)


# Calculate how well the regression line is predicting our values
r_squared = coefficient_of_determination(ys, regression_line)
print("r squared: ", r_squared)

# Visualise data and regression line
plt.scatter(xs, ys, color='#003F72', label='data')
plt.scatter(predict_x, predict_y, label='predicted')
plt.plot(xs, regression_line, label='regression line')
plt.legend(loc=4)
plt.show()
```

Appendix C

K Nearest Neighbors

In this appendix you can find the code for a K nearest neighbors implementation using *sklearn*.

```
import numpy as np
from sklearn import preprocessing, cross_validation, neighbors
import pandas as pd

# Read our breast cancer data, gathered from UCI:
#         https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)
df = pd.read_csv('data/breast-cancer-wisconsin.data')
# Replace the missing values (represented as ?) by -99999
df.replace('?', -99999, inplace=True)
# The ID column is not a good classifier, so we will drop that column
df.drop(['id'], 1, inplace=True)

# Define our features (every column except for the class column)
X = np.array(df.drop(['class'], 1))
# Define our labels (the class column)
y = np.array(df['class'])

# Create training and testing data
X_train, X_test, y_train, y_test = cross_validation.train_test_split(X, y, test_size=

# Define the classifier, a k nearest neighbor classifier
clf = neighbors.KNeighborsClassifier()

# Train the classifier
clf.fit(X_train, y_train)

# Check the accuracy of our trained model
accuracy = clf.score(X_test, y_test)
# Show us the accuracy
print("Accuracy:", accuracy)

# Let's predict something
# Our random to predict features:
example_measure = np.array([4, 2, 1, 1, 2, 3, 2, 1]).reshape(1, -1)
# Predict the result for our random sample
prediction = clf.predict(example_measure)
# Show us the prediction
print("prediction:", prediction)
```

Appendix D

Manual K Nearest Neighbors

In this appendix you find a linear K nearest neighbors build from the ground up.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
# We will use warnings to warn about using lower number of K's than we have groups
import warnings
# To count the votes
from collections import Counter
import pandas as pd
import random

# Set the plot style
style.use('fivethirtyeight')

# Create some random data
dataset = {'k': [[1, 2], [2, 3], [3, 1]], 'r': [[6, 5], [7, 7], [8, 6]]}
# The features of the example we want to classify in k or r
new_features = [5, 7]


# A function that will return the k nearest neighbors to a given point
# @param data:
#       a dictionary containing the classes and the data for those classes
# @param predict:
#       a vector with the features where for we will make a class prediction
# @param k:
#       the number of nearest neighbors to return, default value 3
# @warning:
#       Throws a warning when the given k is <= the number of elements in the given d
def k_nearest_neighbors(data, predict, k=3):
    # First create a warning when the number of data points is smaller than or equal
    if len(data) >= k:
        warnings.warn('K␣is␣set␣to␣a␣value␣less␣than␣total␣voting␣groups!')

    # List with all points and there distances to the prediction
    distances = []
    # For every group calculate the euclidean distance per feature and put it in the
    for group in data:
        for features in data[group]:
            # Calculating the Euclidean Norm, we are using numpy because the calculati
            # than when we would do it manually
            euclidean_distance = np.linalg.norm(np.array(features) - np.array(predict
            distances.append([euclidean_distance, group])
    # Sort the distances and take the first k elements
    votes = [i[1] for i in sorted(distances)[:k]]
    # Count the votes
    # 1 is the number you want, it returns a list of elements like ('r', 3) with 'r'
    # number of votes, so we take the first element and then the class name by doing
    vote_result = Counter(votes).most_common(1)[0][0]
    return vote_result

# Use the k nearest neighbor algorithm to predict the color of the new_features
result = k_nearest_neighbors(dataset, new_features)
# Show us the resulting color
print(result)

# Show our current data
# First manipulate the data a bit and add it in a scatter plot, very nice line btw
[[plt.scatter(ii[0], ii[1], s=100, color=i) for ii in dataset[i]] for i in dataset]
# Throw in the example we want to predict, show the resulting color as well
plt.scatter(new_features[0], new_features[1], s=100, color=result)


# Let's now look at the accuracy on the breast cancer data

# Read our breast cancer data, gathered from UCI:
#       https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)
df = pd.read_csv('data/breast-cancer-wisconsin.data')
# Replace the missing values (represented as ?) by -99999
df.replace('?', -99999, inplace=True)
# The ID column is not a good classifier, so we will drop that column
df.drop(['id'], 1, inplace=True)
# Converting the entire data frame to floats
full_data = df.astype(float).values.tolist()

# Shuffle the data
random.shuffle(full_data)
# Split the training and testing data
# Define the test size first
test_size = 0.2
# Define the dictionaries for our test and training data, 2 = benign tumor, 4 = malig
train_set = {2: [], 4: []}
test_set = {2: [], 4: []}
# Split the data in test and training data
train_data = full_data[:-int(test_size*len(full_data))]
test_data = full_data[-int(test_size*len(full_data)):]
# Populate the dictionaries
for i in train_data:
    train_set[i[-1]].append(i[:-1])
for i in test_data:
    test_set[i[-1]].append(i[:-1])

# Train and test the data
# Initialise the total correct predictions to 0 and the total predictions to 0 as wel
correct = 0
total = 0

# For every group in our test_set make a prediction using our train_set
for group in test_set:
    for data in test_set[group]:
        # Vote using the default number of k's as defined in Scikit
        vote = k_nearest_neighbors(train_set, data, k=5)
        # If prediction correct, count is as correct
        if group == vote:
            correct += 1
        # Count total
        total += 1

# Show the accuracy result
print('Accuracy', correct/total)


# Show the plot on the initial basic data set
plt.show()
```