KU Leuven

Machine Learning: A brief Overview

Tom De Groote

February 12, 2017

Contents

Chapter 1

Introduction

1.1 Acknowledgements

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: $m = \frac{\overline{x} \cdot \overline{y} - \overline{xy}}{(\overline{x})^2 - \overline{x^2}}$ 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 6.

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.

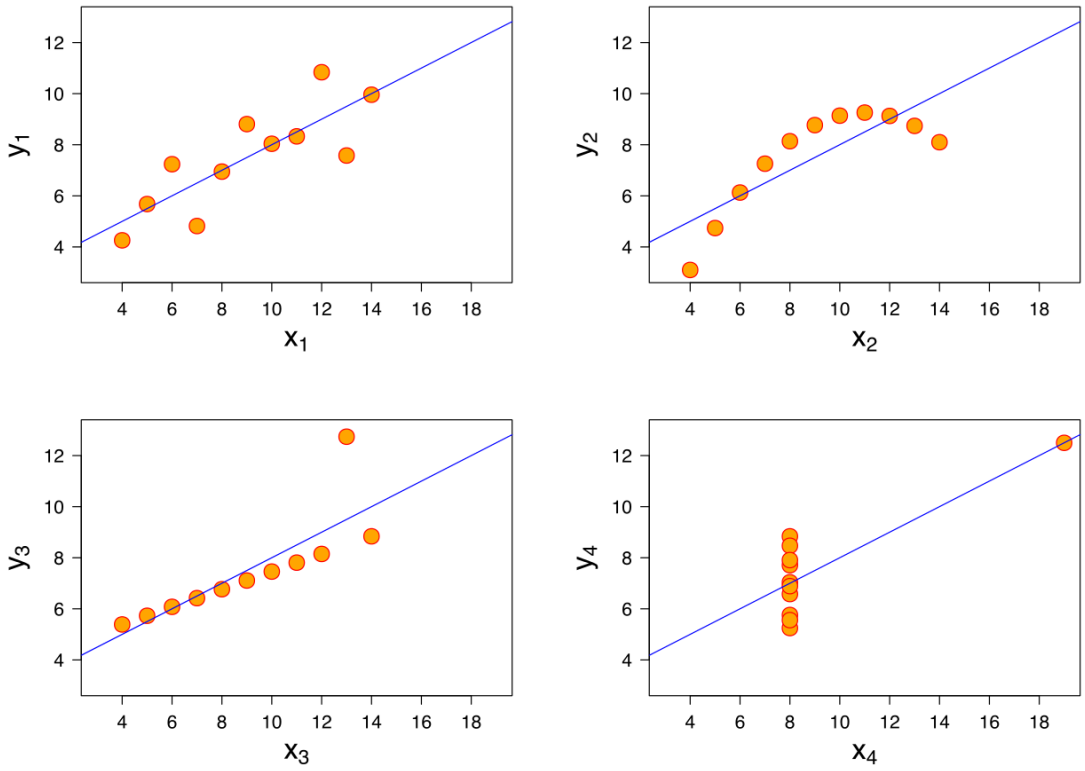

Figure 2.1: Four examples of the function found by linear regression based on the given data points.

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 with 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

KNN 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 for example based on their distance.

A known pitfall for the KNN is that it needs to compare the data in question to all of the points from the dataset. Therefor accuracy is easy to accomplish, but being fast is hard. A way to make it faster is to compare your data only to data within a certain radius. Other pitfalls include: problems with outliers and bad data.

To see how well the KNN performs, one can check it's confidence in two ways, as listed below.

- 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 binary classifier. The objective of the Support Vector Machine is to find the best splitting boundary between data. It is a maximum-margin-classifier. It deals in vector space, thus the seperation is done by using a hyperplane. The best hyperplane is the one that contains the widest margin between support vectors, and is called the decision boundary. It is generally much faster than the KNN algorithm and also more resistent for outliers and pointless data.
Steps to find the decision boundary:

1. Find the support vectors, see figure 3.1. We find these support vectors by maximising the distance between all examples of the two classes.
2. The decision boundary runs through the middle of these support vectors, see figure 3.2.

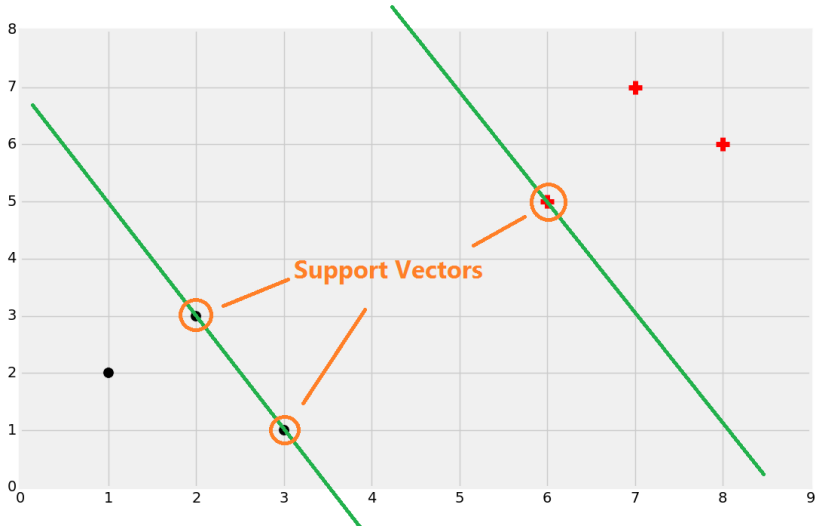

Figure 3.1: Shows the support vectors for this SVM classification problem.

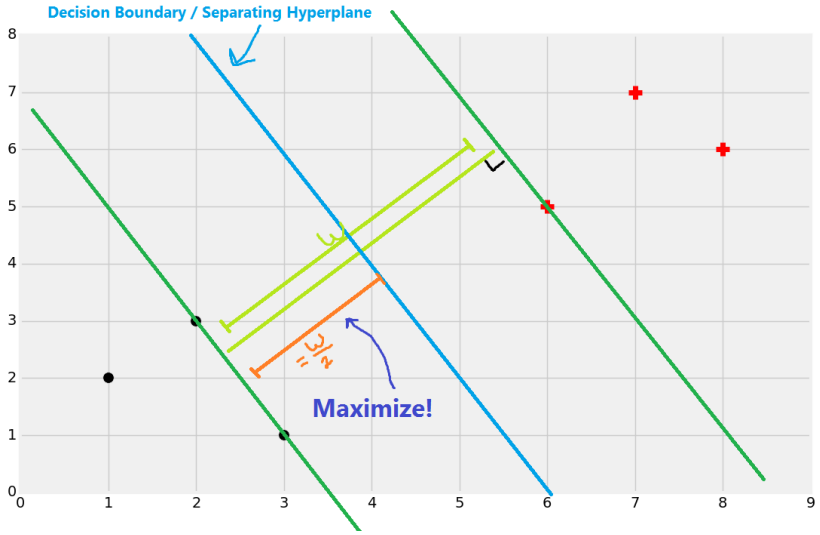

Figure 3.2: Shows the decision boundary for this SVM classification problem.

As you may notice, this method will only work natively on linearly-seperable data and data with only two classes. A way to go beyond the linearly-seperable data limit is using kernels as explained in subsection 3.2.1. To be able to do 3+ classification you can use OVR or OVO, see section 3.2.3 For more details about the specific workings of SVMs I'd like to point out a great SVM Tutorial[1].

3.2.1 Kernels

Kernels are similarity functions, which take two inputs and return a similarity using inner products. This allows us to translate our data to a plausibly infinite number of dimensions in order to find one that has linear separbility, without paying the processing costs to do it. The only requirement to use kernels is to confirm that every interaction with our featurespace is an inner product. The formulas used for SVMs are transformable to inner products, so that is nice. Kernels are also used in other machine learning algorithms than SVMs.

[1] http://www.svm-tutorial.com/

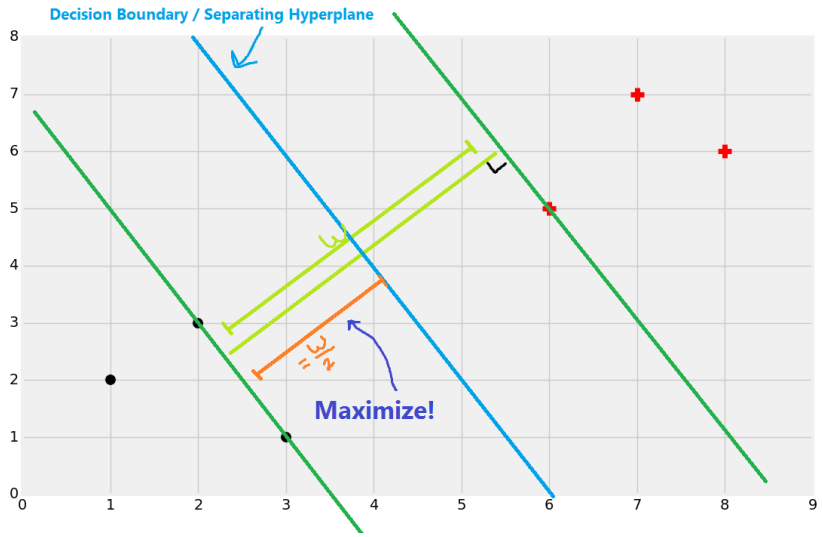

Figure 3.3: Shows the results of van OVR classification.

3.2.2 Soft Margin SVMs

A soft margin SVM allows for some slack on the errors that we might get in the optimisation process (While the standard SVM, a hard-margin, does not allow this wiggle room for error). You can use soft margin SVMs when your data is not perfectly linearly separable, but is very close or when you have a strong over-fitment when using a hard-margin SVM. An over-fitment can be recognised by the number of points on your support vectors versus the number of points of that class. For example if 100% of the positive class' points are on the support vector, this signals a high chance of over-fitment.

3.2.3 OVR and OVO

OVR or *One Verse Rest* seperates each group from the rest and generates decision boundaries for all of these. For example if you have three classes, 1, 2 and 3. Then OVR would compare 1 to (2 and 3), 2 to (1 and 3) and 3 to (1 and 2) and generate decision boundaries for all three options. The problem is that you will almost always have more negatives than positives, since you're maybe comparing one group to three others. This means every classification boundary is actually unbalanced.

OVO or *One Verse One* compares every one class to all others. Thus 1 to 2, 1 to 3, 2 to 1, 2 to 3, 3 to 1 and 3 to 2 deliver all seperate decision boundaries. This tends to result in more balance results.

3.2.4 Code examples

Two code approaches have been made. The first approach uses the *sklearn* kit, the approach can be found in appendix E and is very similar to appendix C. The second approach shows a more basic SVM 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 F. No detailed implementation of kernels, soft margin SVMs, OVO or OVR has been implemented.

Chapter 4

Clustering

Clustering is a form of unsupervised learning and comes in two major forms: Flat and Hierarchical. With both forms the machine is tasked with receiving a dataset that is just featuresets, and then the machine searches for groups and assigns classes on its own. With Flat clustering the scientist tells the machine how many classes/clusters to fine. With Hierarchical clustering, the machine figures out the groups and how many.

The objective is to find relationships and meaninig in data. It can be used in a semi-supervised context, where the scientists use the results from clustiring for further classification or it can be used on truly unknown data, in an attempt to find some structure. Clustering can also be used for typical classificaton, you just don't need to actually feed it what the classifications are beforehand.

4.1 K-Means

K-Means tries to cluster a given dataset into K clusters. Since you have to give it the number of required clusters, it is a flat clustering algorithim. It works as follows:

1. Take the entire dataset, and set, randomly, K number of centroids.
2. Calculate the distance of every featureset to the centroids, and classify based on the nearest centroid.
3. Now take the mean of all groups, set these as the K centroids and repeat untill optimised. Optimisation is often measured as movement of the centroid.

4.1.1 Code examples

Three code approaches have been made. The first approach uses the *sklearn* kit's K-Means classifier to classify a simple example, the approach can be found in appendix G. The second approach also uses the *sklearn* kit's K-Means classifier but this time to classify nonnumerical data, this approach can be found in appendix H. The third and final approach shows a more basic K-Means classifier which illustrates it's fundamentals. This code can help you understand the basic building blocks of the K-Means classifier. This code can be found in appendix I.

4.2 Mean Shift

Mean Shift is very similar to the K-Means algorithm but you do not need to give it the number of clusters in advance, therefor it is a hierarchical clustering algorithm. The Mean Shift algorithm thus follows the following steps:

1. Make all datapoints centroids
2. Take mean of all featuresets within a centroid's radius, setting this mean as new centroid.
3. Repeat step #2 until convergence.

As you may notice the downside is scalability, because we have to start from every datapoint. For finding the mean in step 2 we can use a kernel, you can either use a flat kernel (here we have every featureset with the same weigth) or a Gaussian Kernel (here weight's are assigned by proximity to the kernel's center). The radius as well can be calculated based on the data, as shown in the code-example L.

4.2.1 Code examples

Three code approaches have been made. The first approach uses the *sklearn* kit's Mean Shift classifier to classify a simple example, the approach can be found in appendix J. The second approach also uses the *sklearn* kit's Mean Shift classifier but this time to classify nonnumerical data, this approach can be found in appendix K. The third and final approach shows a more basic Mean Shift classifier which illustrates it's fundamentals. This code can help you understand the basic building blocks of the Mean Shift classifier. This code can be found in appendix L.

Chapter 5

Neural Network

An artificial neural network network mimics the working of a real brain. The brain exists of neurons that transmit information via synapseses. The computer science way to mimic these neurons is shown in figure 5.1. These work in the bigger context as shown in figure 5.2. In this figure you see different kinds of layers, the input layer (which handles the input), the hidden layer(s) (which pass information) and the output layer (the results). To go from one layer to next you weight the input data, and pass it through the function in the neuron. The function in the neuron is a threshold function (aka activiation function). It is basically the sum of all of the weighted values above or below a certain value. If it is above the threshold the neuron fires a signal (1) otherwise it fires (0). This in turn is then weighted and passed to the next neuron untill the output layer is reached. To optimise we can actually drop the threshhold value (thanks to Paul Werbos) and use a sigmoid function instead for making the decisions. The weights are initially chosen at random and will be optimised later on.

In a typical feed forward neural network you first pass the information through the network and then you compare your output layer to the hoped results. From here you begin adjusting the weights to minimise loss/cost, this is called backpropagation. This is an optimisation problem that can be used using for example stochastic gradient descent (or more recent options like AdaGrad and the Adam Optimiser). Once all your examples (and you need a lot of examples) are run and your network is optimised, you can start using this network to make often very good predictions.

Sometimes dropouts are used to mimic dead neurons in the brain. This decreases the chance of over-weighted, or otherwise biasing, neurons in the artificial neural network.

5.1 Feed Forward Neural Networks

A feed forward neural network simply passes the information straight through the neural net and uses backpropagation to optimise the weights. For a sentiment predictor Neural Net implementation look at appendix R and for an example implementation for number recognition take a look at appendix Q.

5.1.1 Code

A simple code example of a Feed Forward Neural Network using TensorFlow can be found in appendix O.

5.2 Recurrent Neural Networks

Addresses the necessity of understanding data in sequences. In RNN the output of the previous cyclus is used in the activiation function of this cyclus.

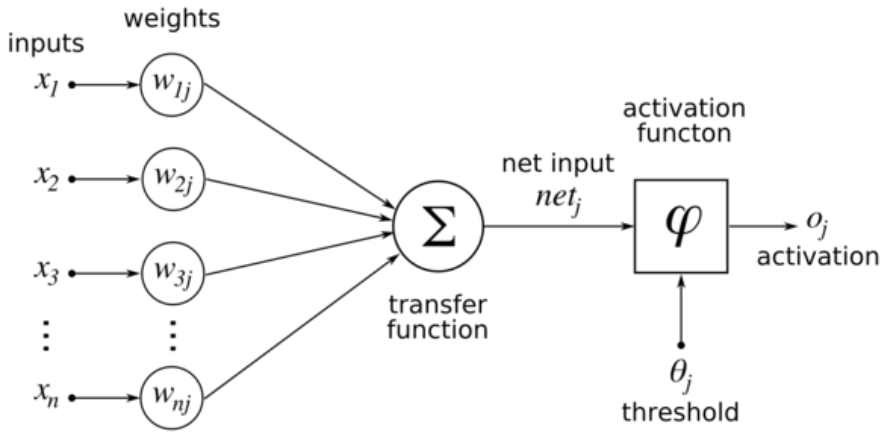

Figure 5.1: Shows the detailed working for one node in an artificial neural network.

5.2.1 LSTM

The Long Short Term Memory cell determines the weight of the previous output and the current input so that your initial input doesn't start to dominate the output. Basically it decides what data to keep and what data to forget. See link[1] for more information.

5.2.2 Code

A simple code example of a LTSM using TensorFlow can be found in appendix P.

5.3 Convolutional Neural Network

Convolution takes the original data and creates feature maps from it. Pooling is the same as down-sampling. Most often used is max-pooling where a region is selected and all values of the region is set equal to the maximum value in that region. The fully connected layer is your typical multilayer perceptron. It roughly has the following structure:

1. Convolution
2. Pooling
3. Convolution
4. Pooling
5. Fully Connected Layer
6. Output

5.3.1 Code

A simple code example of a Convolutional Neural Network using TensorFlow can be found in appendix N. For an implementation of the same code using TFLearn take a look at appendix S

[1] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

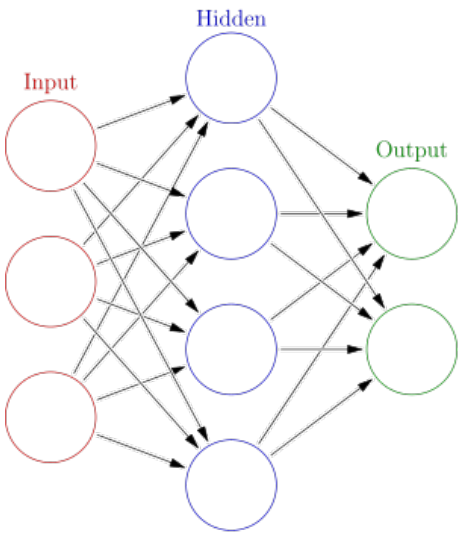

Figure 5.2: Shows the general overview of an artificial neural network.

Chapter 6

General Terms

Backpropagation

Centroids Centroids are the centers of clusters.

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

Convex

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.

Deterministic Enviroment The endstate of the enviroment can be determined based on the current state of the enviroments and its components.

Decision Boundary In a statistical-classification problem with two classes, a decision boundary is a hyperplane that partitions the underlying vector space into two sets, one for each class.

Dot Product Also known as *scalair product* or *inner product* of two vectors $\vec{A}$ and $\vec{B}$ is defined as follows: $\vec{A} \cdot \vec{B} = a_1b_1 + a_2b_2 + ... + a_nb_n$. Two vectors are perpindicular if their dot product equals 0. Dutch translation: *inwendig product*

Down-sampling

Epoch A cycle of feed forward and back propagation in feed forward neural networks. Too many epochs can cause overfitment, too few can cause bad results.

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.

Gradient Descent

Hyperplane A hyperplane is a subspace of one dimension less than its ambient space. If a space is 3-dimensional then its hyperplanes are the 2-dimensional planes.

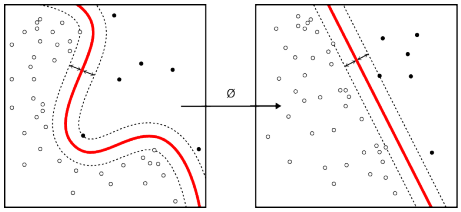

Figure 6.1: An example of simplifying the data by increasing it's dimension.

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 6.1.

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

Lemmatizer A lemmatizertakes similar words and converts them into the same single word.

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

(Maximum) Margin Classifier A margin classifier is a classifier which is able to give an associated distance from the decision boundary for each example. A maximum margin classifier maximises the distance between the decision boundary and all examples.

Object Oriented Programming In short OOP makes it possible to make objects with attributes, these objects can have a certain link towards eachother (subclasses and superclasses).

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.

Machine Learning Classifier

Machine Learning Model

Norm

One Hot A term from electronics where just one element, out of the others is on. This is useful for multiclass classification tasks. For example representing numbers 0 to 9:

- $0 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
- $1 = [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$
- $2 = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$

  ...

- $9 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]$

Overfitting Overfitting is the act of fitting the function too much to the given sample data, a bad consequence of overfitting is that future data will not fit the calculated function. And example is shown in figure 6.2.

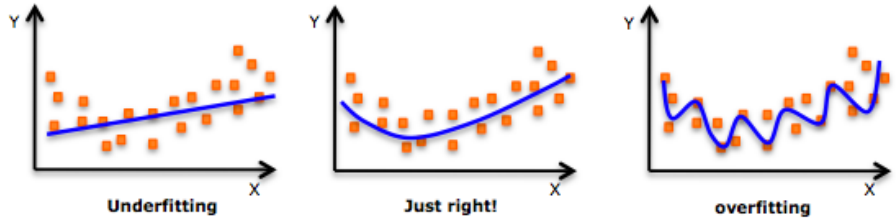

Figure 6.2: Shows underfitting, right fitting and overfitting.

Sigmoid Function

Stochastic Enviroment The endstate of the enviroment can not be exactly determined based on the current state of the enviroment and its objects since there is randomness involved.

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.

Support Vector The points closest to the maximum-margin-hyperplane, as shown in figure 3.1.

Tensor A tensor is an array-like object that can hold a matrix, vector or even just a scalar.

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.

Tokenizer A tokenizer sepertates elements. For example a word tokenizer seperates words.

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.

Unsupervised Learning As opposed to supervised learning, the scientist doesn't tell the machine what the classes of featuresets were.

Vector A vector has a *magnitude* and a *direction*. The *magnitude* is the same as the Euclidean distance or norm. For example the $\vec{A} = [4, 3]$ has the direction 4 in dimension 1 and 2 in dimension 2, the magnitudes is $\sqrt{4^2 + 3^2} = 5$.

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' i
    # 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 = malign
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()
```

Appendix E

Support Vector Machine

In this appendix you can find the code for a Support Vector Machine implementation using *sklearn*.

```
import numpy as np
from sklearn import preprocessing, cross_validation, svm
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 support vector machine
clf = svm.SVC()

# 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 F

Manual Support Vector Machine

In this appendix you find a Support Vector Machine build from the ground up.

```
import matplotlib.pyplot as plt
from matplotlib import style
import numpy as np

# Our plot style
style.use('ggplot')

# Our starting data
# As you can see we have two classes, -1 and 1 with some data for those classes
data_dict = {-1: np.array([[1, 7], [2, 8], [3, 8], ]),
              1: np.array([[5, 1], [6, -1], [7, 3], ])}


# A class definition of a SVM
# The class doesn't explain all the details of how an SVM works, for more information
# look at svm-tutorial.com
class SupportVectorMachine:

    # Runs whenever an object is created
    def __init__(self, visualisation=True):
        # Set the object variable visualisation to the given value
        self.visualisation = visualisation
        # Set the colors for the two classes
        self.colors = {1: 'r', -1: 'b'}

        # Visualise if necessary
        if self.visualisation:
            self.fig = plt.figure()
            self.ax = self.fig.add_subplot(1, 1, 1)

    # A method that visualises the SVM
    def visualise(self):
        # scattering known featuresets
        [[self.ax.scatter(x[0], x[1], s=100, color=self.colors[i]) for x in data_dict

        # Returns the hyperplane point given x, w, b and v
        def hyperplane(x, w, b, v):
            return (-w[0] * x - b + v) / w[1]

        # Some useful variables
        datarange = (self.min_feature_value*0.9, self.max_feature_value*1.1)
        hyp_x_min = datarange[0]
```

```
        hyp_x_max = datarange[1]

        # Graph the positive support vector hyperplane
        # w.x + b = 1
        psv1 = hyperplane(hyp_x_min, self.w, self.b, 1)
        psv2 = hyperplane(hyp_x_max, self.w, self.b, 1)
        self.ax.plot([hyp_x_min, hyp_x_max], [psv1, psv2], "k")

        # Graph the negative support vector hyperplane
        # w.x + b = -1
        nsv1 = hyperplane(hyp_x_min, self.w, self.b, -1)
        nsv2 = hyperplane(hyp_x_max, self.w, self.b, -1)
        self.ax.plot([hyp_x_min, hyp_x_max], [nsv1, nsv2], "k")

        # Graph the decision surface
        # w.x + b = 0
        ds1 = hyperplane(hyp_x_min, self.w, self.b, 0)
        ds2 = hyperplane(hyp_x_max, self.w, self.b, 0)
        self.ax.plot([hyp_x_min, hyp_x_max], [ds1, ds2], "k")

        # show the plot
        plt.show()

    # Predict the class given a set of features
    def predict(self, features):
        # Find the classification given a set of features by looking at the sign of th
        # following formula: sign(x.w +b)
        classification = np.sign(np.dot(np.array(features), self.w) + self.b)

        # For visualisation puposes:
        if classification != 0 and self.visualisation:
            self.ax.scatter(features[0], features[1], s=200, marker='*', c=self.colors
        elif self.visualisation:
            print('featureset', features, 'is on the decision boundary')

        return classification

    # train the SVM
    def fit(self, data):
        self.data = data

        # Begin building an optimisation dictionary, which contains any optimisation
        # When all optimisations are done we will pick the [w, b] value with the lowe
        # in this dictionary
        #
        # The optimisation is done by stepping down the vector w and calculating the
        # fit the x.w + b equation. The found values will be saved in this dictionary
        #
        # How the dictionary will look: {||w||: [w, b]}
        opt_dict = {}

        # This enables us to check every version of the vector possible
        transforms = [[1, 1], [-1, 1], [-1, -1], [1, -1]]

        # Pick a starting point that fits our data
        # First put all features in all_data
        all_data = []
        for yi in self.data:
```

```
        for featureset in self.data[yi]:
            for feature in featureset:
                all_data.append(feature)

    # Then pick the starting point
    self.max_feature_value = max(all_data)
    self.min_feature_value = min(all_data)

    # We don't need all data anymore so can throw it out of memory
    all_data = None

    # For support vectors yi(xi.w + b) = 1, we will now start looking for this by
    # bill

    # Pick step_sizes for stepping down the vector, we start with big steps,
    # when find the minimum for this step size, we start taking smaller steps to
    # By not starting with the smallest steps we save a lot of calculating power.
    # This is one way to approach the optimisation problem. With these steps we w
    step_sizes = [self.max_feature_value * 0.1,
                  self.max_feature_value * 0.01,
                  self.max_feature_value * 0.001]

    # With these steps we will approach b
    # You could implement a similar step size feature for finding an accurate b th
    # But for brevity this was skipped. (Might be a TODO for later)
    b_range_multiple = 5
    b_multiple = 5
    latest_optimum = self.max_feature_value*10

    # Now we are ready to approach w and b by stepping
    for step in step_sizes:
        # Initial pick for w, we can do this because it is a convex problem
        w = np.array([latest_optimum, latest_optimum])

        # Step down the convex bowl until you find the optimal point with this ste
        optimized = False
        while not optimized:
            # Iterate through possible b values
            # Try every b from -max_feature_value * b_range_multiple to +... with
            # step * b_multiple
            for b in np.arange(-1*(self.max_feature_value*b_range_multiple),
                               self.max_feature_value*b_range_multiple,
                               step*b_multiple):
                # Check the equation x.w + b for every possible transformation
                for transformation in transforms:
                    # Transform w
                    w_t = w * transformation
                    #
                    found_option = True
                    # Implement the constraint: yi(xi.w + b) >= 1
                    # Weakest link of SVM, SMO tries to fix this
                    for i in self.data:
                        for xi in self.data[i]:
                            yi = i
                            # Check if constraint fulfilled
                            if not yi*(np.dot(w_t, xi) + b) >= 1:
                                found_option = False
                                break  # TODO als het niet werkt kom eens hier ki
```

```
                            if not found_option:
                                break  # TODO als het niet werkt kom eens hier kijken
                        # If constrained fulfilled then add the option to our optimis
                        if found_option:
                            opt_dict[np.linalg.norm(w_t)] = [w_t, b]

                    # Check if the first element of the vector w is < 0,  we can stop sea
                    # already check the w's < 0  with the transformations
                    if w[0] < 0:
                        optimized = True
                        print('Optimized a step.')
                    else:
                        # Step on w to approximate the optimal value better
                        w = w - step

                # We found an optimum, so let's see what our best possible w and b are. F
                # The w and b where the magnitude (aka the norm) of w is minimal
                # Calculate all the norms
                norms = sorted([n for n in opt_dict])
                # Our optimal choice
                opt_choice = opt_dict[norms[0]]
                self.w = opt_choice[0]
                self.b = opt_choice[1]
                # Redefine our latest optimum
                latest_optimum = opt_choice[0][0] + step*2


# Create an SVM object
svm = SupportVectorMachine()
# Make our data fit
svm.fit(data=data_dict)

# Values to predict
predict_us = [[0,10],
              [1,3],
              [3,4],
              [3,5],
              [5,5],
              [5,6],
              [6,-5],
              [5,8]]

# Make predictions
for p in predict_us:
    svm.predict(p)

# Show us the magic
svm.visualise()
```

Appendix G

K-Means

In this appendix you find the use of *scikit*'s K-Means classifier for simple data.

```
import matplotlib.pyplot as plt
from matplotlib import style
import numpy as np
from sklearn.cluster import KMeans

style.use('ggplot')

# Our data set
X = np.array([[1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [9, 11]])

# Fit the data
clf = KMeans(n_clusters=2)
clf.fit(X)

# Grab the centroids
centroids = clf.cluster_centers_
# Grab the labels
labels = clf.labels_

# visualise
colors = ["g.", "r.", "c.", "y."]
# Color the data set according to the given labels by the classifier
for i in range(len(X)):
    plt.plot(X[i][0], X[i][1], colors[labels[i]], markersize=10)
# Also show the centroids
plt.scatter(centroids[:, 0], centroids[:, 1], marker="x", s=150, linewidths=5, zorder=
plt.show()
```

Appendix H

Nonnumerical K-Means

In this appendix you find the use of *scikit*'s K-Means classifier for nonnumerical data.

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn import preprocessing, cross_validation
import pandas as pd

# Read the excel file which contains the data
df = pd.read_excel('../data/titanic.xls')
# Drop unimportant columns
df.drop(['body', 'name'], 1, inplace=True)
df.fillna(0, inplace=True)


# Convert the non-numerical data in the data set to numerical data
def handle_non_numerical_data(df):
    columns = df.columns.values
    # Go through every column
    for column in columns:
        # Initialise a dictionary for our numerical value of a label
        text_digit_vals = {}

        # Function to convert a textual label to its numerical representation
        def convert_to_int(val):
            return text_digit_vals[val]

        # If the values of a column are nonnumerical, convert them
        if df[column].dtype != np.int64 and df[column].dtype != np.float64:
            # The contents of a column
            column_contents = df[column].values.tolist()
            # The unique elements of a column
            unique_elements = set(column_contents)
            # Add all unique elements to the dictionary
            x = 0
            for unique in unique_elements:
                if unique not in text_digit_vals:
                    text_digit_vals[unique] = x
                    x += 1
            # Convert column from their nonnumerical representation to their numerica
            df[column] = list(map(convert_to_int, df[column]))
    return df

# Convert the nonnumerical data to numerical data
df = handle_non_numerical_data(df)
```

```
# Since using flat clustering we need to tell it how many clusters to find.
# We are going to look for groups of survivors and non-survivors

# If you want to see how much of an impact individual features have on the end result
# You can uncomment these lines. If the accuracy drops significantly,
# it has a large impact on the results.
# df.drop(['boat'], 1, inplace=True)
# df.drop(['sex'], 1, inplace=True)

# Our X data (without the survivor feature)
X = np.array(df.drop(['survived'], 1).astype(float))
# Preprocess our X, aka scale it to range between -1 and 1
X = preprocessing.scale(X)
# Our y data, aka the survivor feature
y = np.array(df['survived'])

# Define our classifier
clf = KMeans(n_clusters=2)
# Fit the data
clf.fit(X)

# Let's check our accuracy, for us survivors are 1 and non-survivors are 0, but cluste
# uses random labels, so our accuracy is whatever is highest, since with clustering ne
# could be labeled 1 and survivors 1.
correct = 0
for i in range(len(X)):
    predict_me = np.array(X[i].astype(float))
    predict_me = predict_me.reshape(-1, len(predict_me))
    prediction = clf.predict(predict_me)
    if prediction[0] == y[i]:
        correct += 1

# Calculate the accuracy of the predictions
accuracy = (correct/len(X))
print("Accuracy:", accuracy)
```

Appendix I

Manual K-Means

In this appendix you find the implementation of a K-Means classifier from the ground up.

```
import matplotlib.pyplot as plt
from matplotlib import style
import numpy as np

style.use('ggplot')

# Our data set
X = np.array([[1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [9, 11]])


class KMeans:
    # @var k
    #       The number of clusters
    # @var tol
    #       The tolerance for optimisation
    # @var max_iter
    #       The maximum number of iterations to run for the optimisation.
    def __init__(self, k=2, tol=0.001, max_iter=300):
        self.k = k
        self.tol = tol
        self.max_iter = max_iter

    # This method fits the given data to the model
    def fit(self, data):
        # Set the initial centroids, to the first k elements of the given data set
        self.centroids = {}
        for i in range(self.k):
            self.centroids[i] = data[i]

        # For a maximum nr of iterations optimise the data classification
        for i in range(self.max_iter):
            # Initialising classification dictionary
            self.classifications = {}
            for j in range(self.k):
                self.classifications[j] = []

            # Do the actual classification based on the distance of the data point to
            for featureset in data:
                # The distance to all centroids of a given feature set
                distances = [np.linalg.norm(featureset-self.centroids[cen]) for cen in
                # Classify the feature set to the class of the centroid with the smal
                classification = distances.index(min(distances))
```

```
                self.classifications[classification].append(featureset)

            # Remember the previous centroids to check if we fulfill the tolerance req
            prev_centroids = dict(self.centroids)
            # Find the new centroids
            for classification in self.classifications:
                # The new centroids are the averages of all the feature sets in the c
                self.centroids[classification] = np.average(self.classifications[class

            # Check if currently optimised
            optimised = True
            for c in self.centroids:
                # Previous centroid
                original_centroid = prev_centroids[c]
                # Current centroid
                current_centroid = self.centroids[c]
                # If it moved more than the tolerance, show how much it moved and set
                if np.sum((current_centroid-original_centroid)/original_centroid*100)
                    optimised = False
            # If we fulfill the requirements, we are optimised and can stop optimising
            if optimised:
                break

    # Predicts the class of the given feature set
    def predict(self, data):
        distances = [np.linalg.norm(data - self.centroids[centroid]) for centroid in
        classification = distances.index(min(distances))
        return classification


# Define our classifier
clf = KMeans()
# Fit our data
clf.fit(X)

# visualise
colors = 10*["g", "r", "c", "b", "k"]
# Add the centroids to the plot
for centroid in clf.centroids:
    plt.scatter(clf.centroids[centroid][0], clf.centroids[centroid][1], marker="o", co

# Add all data to the plot, with their classification
for classification in clf.classifications:
    color = colors[classification]
    for featureset in clf.classifications[classification]:
        plt.scatter(featureset[0], featureset[1], marker="x", color=color, s=150, line

# Let's see how well we perform for unknown data
unknowns = np.array([[1, 3], [8, 9], [0, 3], [5, 4], [6, 4]])
# ask a prediction for the unknowns and plot them
for unknown in unknowns:
    classification = clf.predict(unknown)
    plt.scatter(unknown[0], unknown[1], color=colors[classification], marker="*", s=1

# plot
plt.show()
```

Appendix J

Mean Shift

In this appendix you find the use of *scikit*'s Mean Shift classifier for simple data.

```
from sklearn.cluster import MeanShift
from sklearn.datasets.samples_generator import make_blobs
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import style

style.use('ggplot')

# The centers for the generated blob data
centers = [[1, 1, 1], [5, 5, 5], [3, 10, 10]]
# Our feature sets, generate data using a generator
X, _ = make_blobs(n_samples=500, centers=centers, cluster_std=1.5)

# Mean Shift classifier
ms = MeanShift()
# Fit the feature sets
ms.fit(X)
# Get the assigned labels
labels = ms.labels_
# Get the cluster centers
cluster_centers = ms.cluster_centers_
sorted(cluster_centers, key=lambda x: x[0])

# Show the predicted centers vs the original ones
print(len(centers), "original centers for data generation:", centers)
print(len(cluster_centers), "predicted centers by Mean Shift:", cluster_centers)

# Initialise the 3D plot
colors = 10*['r', 'g', 'b', 'c', 'k', 'y', 'm']
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Add the feature sets to the plot
for i in range(len(X)):
    ax.scatter(X[i][0], X[i][1], X[i][2], c=colors[labels[i]], marker='o')

# Add the cluster centers to the plot
ax.scatter(cluster_centers[:, 0], cluster_centers[:, 1], cluster_centers[:, 2], marke
           linewidths=5, zorder=10)

# Show the plot
plt.show()
```

Appendix K

Nonnumerical Mean Shift

In this appendix you find the use of *scikit*'s Mean Shift classifier for nonnumerical data.

```
import numpy as np
import pandas as pd
from sklearn.cluster import MeanShift
from sklearn import preprocessing

# When running you can ignore the warning, the program takes a while to make it's calc
# Read the excel file which contains the data
df = pd.read_excel('../data/titanic.xls')
original_df = pd.DataFrame.copy(df)
# Drop unimportant columns
df.drop(['body', 'name'], 1, inplace=True)
df.fillna(0, inplace=True)


# Convert the non-numerical data in the data set to numerical data
def handle_non_numerical_data(df):
    columns = df.columns.values
    # Go through every column
    for column in columns:
        # Initialise a dictionary for our numerical value of a label
        text_digit_vals = {}

        # Function to convert a textual label to its numerical representation
        def convert_to_int(val):
            return text_digit_vals[val]

        # If the values of a column are nonnumerical, convert them
        if df[column].dtype != np.int64 and df[column].dtype != np.float64:
            # The contents of a column
            column_contents = df[column].values.tolist()
            # The unique elements of a column
            unique_elements = set(column_contents)
            # Add all unique elements to the dictionary
            x = 0
            for unique in unique_elements:
                if unique not in text_digit_vals:
                    text_digit_vals[unique] = x
                    x += 1
            # Convert column from their nonnumerical representation to their numerica
            df[column] = list(map(convert_to_int, df[column]))
    return df
```

```
# Convert the nonnumerical data to numerical data
df = handle_non_numerical_data(df)

# Our X data (without the survivor feature)
X = np.array(df.drop(['survived'], 1).astype(float))
# Pre-process our feature sets, aka scale it to range between -1 and 1
X = preprocessing.scale(X)
# Our y data, aka the survivor feature
y = np.array(df['survived'])

# Define our classifier
clf = MeanShift()
# Fit the data
clf.fit(X)

# Get the labels from our classifier
labels = clf.labels_
# Get the cluster centers from our classifier
cluster_centers = clf.cluster_centers_

# Add a cluster group column to our original data set (This is to view what kind of c
original_df['cluster_group'] = np.nan
# populate the new column
for i in range(len(X)):
    original_df['cluster_group'].iloc[i] = labels[i]

# Check the survival rate of each of the found groups
# First get the number of clusters
n_clusters_ = len(np.unique(labels))
# initialise a dictionary for the survival_rates of the clusters
survival_rates = {}
# Add the survival rates of the clusters to the dictionary
for i in range(n_clusters_):
    # Get all the rows in our original df where the cluster group equals the current
    temp_df = original_df[(original_df['cluster_group'] == float(i))]
    # Get all the survivals in the cluster
    survival_cluster = temp_df[(temp_df['survived'] == 1)]
    # Calcualte survival rate
    survival_rate = len(survival_cluster)/len(temp_df)
    survival_rates[i] = survival_rate

# Show us the survival rates, the number of groups and survival rates can differ ever
# randomness is involved
print(survival_rates)
```

Appendix L

Manual Mean Shift

In this appendix you find the implementation of a Mean Shift classifier from the ground up.

```
import matplotlib.pyplot as plt
from matplotlib import style
import numpy as np

style.use('ggplot')

# Our data set
X = np.array([[1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [9, 11], [8, 2], [10, 2],

# Used colors
colors = 10*["g", "r", "c", "b", "k"]


# The Mean Shift class
class MeanShift:

    # @var radius
    #       The radius for our Mean Shift algorithm, our algorithm will find this by
    # @var radius_norm_step
    #       The step with which the radius increases.
    def __init__(self, radius=None, radius_norm_step=100):
        self.radius = radius
        self.radius_norm_step = radius_norm_step

    # Fits the given data to the classifier
    def fit(self, data):
        # If the radius is not defined, we will find it ourselves
        if not self.radius:
            all_data_centroid = np.average(data, axis=0)
            all_data_norm = np.linalg.norm(all_data_centroid)
            self.radius = all_data_norm / self.radius_norm_step

        # Initialise a dictionary for our centroids
        centroids = {}

        # the initial centroids are all our data points
        for i in range(len(data)):
            centroids[i] = data[i]

        # our weights for changing the radius
        weights = [i for i in range(self.radius_norm_step)][::-1]
        # Optimise the centroids
```

```
        while True:
            new_centroids = []
            # For every centroid, find every all feature sets within it's radius and
            # centroid based on the mean of these feature sets.
            for i in centroids:
                in_radius = []
                centroid = centroids[i]
                # Go over every feature set to see if it's in the current centroid's
                # Let every feature set vote for this change in radius, by using thei
                for featureset in data:
                    distance = np.linalg.norm(featureset-centroid)
                    if distance == 0:
                        distance = 0.0000000001
                    weight_index = int(distance/self.radius)
                    if weight_index > self.radius_norm_step-1:
                        weight_index = self.radius_norm_step-1

                    to_add = (weights[weight_index]**2)*[featureset]
                    in_radius += to_add

                # Calculate the new centroid based on all feature sets in the radius
                new_centroid = np.average(in_radius, axis=0)
                # and add it as a tuple to the new centroids
                new_centroids.append(tuple(new_centroid))
            # Continue only with the unique new centroids
            uniques = sorted(list(set(new_centroids)))

            # Because our radius is changing, it is possible that centroids are very
            # We want to merge these:
            to_pop = []
            for i in uniques:
                for ii in [i for i in uniques]:
                    if i == ii:
                        pass
                    # Check if the given centroid is within the radius of another cen
                    elif np.linalg.norm(np.array(i)-np.array(ii)) <= self.radius:
                        to_pop.append(ii)
                        break
            # Remove the centroids that are too close to other centroids
            for i in to_pop:
                try:
                    uniques.remove(i)
                except:
                    pass

            # Remember the previous centroids
            prev_centroids = dict(centroids)

            # Set the unique new centroids as the current centroids
            centroids = {}
            for i in range(len(uniques)):
                centroids[i] = np.array(uniques[i])

            # Check if optimised, only optimised when the current centroids are equal
            optimised = True
            for i in centroids:
                if not np.array_equal(centroids[i], prev_centroids[i]):
                    optimised = False
```

```
                if not optimised:
                    break

            # If optimised, break
            if optimised:
                break

        # Set the centroids of this class to the calculated centroids
        self.centroids = centroids

        # Classify our data as well on the found centroids
        self.classifications = {}

        # Add a place in the dictionary for every centroid
        for i in range(len(self.centroids)):
            self.classifications[i] = []

        # Classify every feature set in the data
        for featureset in data:
            # Compare distance to all centroids
            distances = [np.linalg.norm(featureset-self.centroids[centroid]) for cent
            # The classification is the centroid that is closest
            classification = (distances.index(min(distances)))

            # add the classification
            self.classifications[classification].append(featureset)

    # Predicts the class of a given feature set
    def predict(self, data):
        # Compare distance to all centroids
        distances = [np.linalg.norm(featureset - self.centroids[centroid]) for centroi
        # The classification is the centroid that is closest
        classification = (distances.index(min(distances)))
        return classification


# Create the classifier
clf = MeanShift()
# Fit the data
clf.fit(X)
# Get our calculated centroids
centroids = clf.centroids

# Visualise
# add the centroids to the plot
for c in centroids:
    plt.scatter(centroids[c][0], centroids[c][1], color='k', marker='*', s=150)

# Add all classifications to the plot
for classification in clf.classifications:
    color = colors[classification]
    for featureset in clf.classifications[classification]:
        plt.scatter(featureset[0], featureset[1], marker="x", color=color, s=150, line
# add the original feature sets to the plot
plt.scatter(X[:, 0], X[:, 1])

# show the plot
plt.show()
```

Appendix M

Create Sentiment Featureset

In this appendix you find the implementation for creating a sentiment featureset.

```
import os
from nltk.tokenize import word_tokenize
import numpy as np
import random
# Pickle helps to save progress, so you don't need to start from the beginning every t
import pickle
from collections import Counter
from nltk.stem import WordNetLemmatizer

# Tp download the natural language toolkit, do:
# import nltk
# nltk.download()

# Define the lemmatizer
lemmatizer = WordNetLemmatizer()
# Set the number of lines used (make smaller if you want a smaller data size)
hm_lines = 100000


# Function that creates a lexicon given a file with positive examples and a file with
def create_lexicon(pos, neg):
    # Initialise the lexicon
    lexicon = []
    with open(pos, 'r') as f:
        # Read all lines in the pos file
        contents = f.readlines()
        # For every line tokenize the words of the line and add them to the lexicon
        for l in contents:
            all_words = word_tokenize(l)
            lexicon += list(all_words)

    # Same principle as for the pos file above
    with open(neg, 'r') as f:
        contents = f.readlines()
        for l in contents:
            all_words = word_tokenize(l)
            lexicon += list(all_words)

    # Lemmatize all words in our lexicon
    lexicon = [lemmatizer.lemmatize(i) for i in lexicon]
    # Count the number of occurrences of every word in our lexicon
    w_counts = Counter(lexicon)
```

```
    # Initialise a new lexicon where we remove very rare and very common words like "
    l2 = []
    for w in w_counts:
        # If a given word shows up more than 50 times but less than a 1000 add them to
        # These values should be tweaked to fit your data set (e.g. based on %) but th
        if 1000 > w_counts[w] > 50:
            l2.append(w)
    # Show the length of this new lexicon
    print("Initial lexicon length", len(lexicon), "Filtered lexicon length:", len(l2))
    return l2


# Function that maps the occurrence of a word in a sample to the zeros array represen
# in a lexicon by turning them "on" (changing them to 1)
def sample_handling(sample, lexicon, classification):
    # Initialise our feature set
    featureset = []

    with open(sample, 'r') as f:
        # Read all lines in the sample file
        contents = f.readlines()
        # Only handle the first hm_line nr of lines
        for l in contents[:hm_lines]:
            # Tokenize the words in the line
            current_words = word_tokenize(l.lower())
            # Lemmatize the words from the line
            current_words = [lemmatizer.lemmatize(i) for i in current_words]
            # Initialize the features with a zer array
            features = np.zeros(len(lexicon))
            # For every word in our lines' words turn the appropriate element in the
            for word in current_words:
                if word.lower() in lexicon:
                    index_value = lexicon.index(word.lower())
                    features[index_value] += 1
            # Add the features of this line to our feature set with the given classifi
            features = list(features)
            featureset.append([features, classification])
    return featureset


# Function to create training and test sets
def create_feature_sets_and_labels(pos, neg, test_size=0.1):
    # Use the create_lexicon function to generate a lexicon for the pos and neg files
    lexicon = create_lexicon(pos, neg)

    # Create features based on all pos and neg features
    features = []
    features += sample_handling(pos, lexicon, [1, 0])
    features += sample_handling(neg, lexicon, [0, 1])
    random.shuffle(features)
    features = np.array(features)

    # Calculate the number of testing examples
    testing_size = int(test_size*len(features))

    # Generate training and testing data sets (x and y)
    train_x = list(features[:, 0][:-testing_size])
```

```
    train_y = list(features[:, 1][:-testing_size])
    test_x = list(features[:, 0][-testing_size:])
    test_y = list(features[:, 1][-testing_size:])

    return train_x, train_y, test_x, test_y


# Runner
if __name__ == '__main__':
    # Our root directory of this file (In the Neural Network folder)
    ROOT_DIR = os.path.dirname(os.path.abspath(__file__))
    train_x, train_y, test_x, test_y = create_feature_sets_and_labels(ROOT_DIR + '/../
                                                                      ROOT_DIR + '/../
    # Write results away with pickle
    with open(ROOT_DIR + '/../pickle_dump/sentiment_set.pickle', 'wb') as f:
        pickle.dump(([train_x, train_y, test_x, test_y]), f)
```

Appendix N

Convolutional Neural Network

In this appendix you find the implementation of a convolutional neural network using TensorFlow.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

# In this file we use the MNIST data set and TensorFlow to predict the number of a giv
# written number from 0 to 9.

# Our data set
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# The number of classes, 0 to 9 so 10 classes
n_classes = 10
# Batch size to train the network
batch_size = 128

# The rate at which we drop neurons
keep_rate = 0.8
keep_prob = tf.placeholder(tf.float32)

# Defining some placeholders in our graph, 784 because we are working on 28 by 28 pix
# the shape of the variable for TensorFlow)
x = tf.placeholder(tf.float32, [None, 784])
y = tf.placeholder(tf.float32, [None, n_classes])


# Convolution
def conv2d(x, W):
    # The stride parameter dictates the movement of the window, in this case one pixe
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')


# Pooling
def maxpool2d(x):
    # The stride parameter dictates the movement of the window, in this case two pixe
    # ksize is the size of the pooling window, in this case a 2x2 window for pooling
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')


def cnn_model(data):
    # Define the weights
    weights = {
        # 5x5 convolution, 1 input image, 32 outputs
        'W_conv1': tf.Variable(tf.random_normal([5, 5, 1, 32])),
```

```
        # 5x5 convolution, 32 input images, 64 outputs
        'W_conv2': tf.Variable(tf.random_normal([5, 5, 32, 64])),
        # Fully connected 7*7*64 inputs, 1024 outputs
        'W_fc': tf.Variable(tf.random_normal([7*7*64, 1024])),
        # Output layer 1024 inputs, 10 outputs (class prediction)
        'out': tf.Variable(tf.random_normal([1024, n_classes]))
    }

    # Define biases
    biases = {
        'b_conv1': tf.Variable(tf.random_normal([32])),
        'b_conv2': tf.Variable(tf.random_normal([64])),
        'b_fc': tf.Variable(tf.random_normal([1024])),
        'out': tf.Variable(tf.random_normal([n_classes]))
    }

    # Reshape input to 4D tensor
    data = tf.reshape(data, shape=[-1, 28, 28, 1])
    # Convolution Layer, using our function conv2d
    conv1 = tf.nn.relu(conv2d(data, weights['W_conv1']) + biases['b_conv1'])
    # Max pooling
    conv1 = maxpool2d(conv1)
    # Convolution layer
    conv2 = tf.nn.relu(conv2d(conv1, weights['W_conv2']) + biases['b_conv2'])
    conv2 = maxpool2d(conv2)
    # Fully connected layer
    fc = tf.reshape(conv2, [-1, 7*7*64])
    fc = tf.nn.relu(tf.matmul(fc, weights['W_fc']) + biases['b_fc'])
    # Dropping some neurons
    fc = tf.nn.dropout(fc, keep_rate)
    # Output layer
    output = tf.matmul(fc, weights['out']) + biases['out']

    return output


def train_neural_network(x):
    # Produces the initial predictions
    prediction = cnn_model(x)
    # Measure how wrong our predictions are
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(prediction, y))
    # We use AdamOptimizer as our cost optimisation function
    optimizer = tf.train.AdamOptimizer().minimize(cost)

    # Define the number of epochs
    hm_epochs = 10
    # Start a session
    with tf.Session() as sess:
        # Initialise all variables
        sess.run(tf.global_variables_initializer())

        # Train for the number of epochs
        for epoch in range(hm_epochs):
            # We will keep track of our error
            epoch_loss = 0
            # Train for every batch
            for _ in range(int(mnist.train.num_examples/batch_size)):
                # get the x and y data for this epoch (from the MNIST data set)
```

```
                epoch_x, epoch_y = mnist.train.next_batch(batch_size)
                # Run the optimiser and cost function on our data
                _, c = sess.run([optimizer, cost], feed_dict={x: epoch_x, y: epoch_y})
                epoch_loss += c
            # Print our progress
            print('Epoch', epoch, 'completed out of', hm_epochs, 'loss', epoch_loss)

        # Count the number of correct predictions
        correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(y, 1))
        # Check our prediction accuracy on the test data set
        accuracy = tf.reduce_mean(tf.cast(correct, 'float'))
        print('Accuracy:', accuracy.eval({x: mnist.test.images, y: mnist.test.labels}))

# Run the training method
train_neural_network(x)
```

Appendix O

Neural Network Example

In this appendix you find the implementation of a simple example of a Neural Network using TensorFlow.

```
# TensorFlow is a package that allows us to work with tensors very efficiently
import tensorflow as tf

# Creates nodes in a graph = the construction phase
# Creates a constant value node equal to 5
x1 = tf.constant(5)
# Creates a constant value node equal to 6
x2 = tf.constant(6)

# Create a multiplication operation node in our graph
result = tf.mul(x1, x2)
# As shown it is still just an abstract graph, the multiplication has not been execute
print("Our abstract graph", result)

# First we build the graph, next we launch it
sess = tf.Session()
# Run the result operation from above
print("The result", sess.run(result))

# Close the session when finished
sess.close()
```

Appendix P

Long Short Term Menomory

In this appendix you find the implementation of a LTSM using TensorFlow.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
from tensorflow.python.ops import rnn, rnn_cell

# This file uses a LSTM to predict the number of a given image. It does this by taking
# in as sequential input as chunks.

# Our data set
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# Some basic variables
hm_epochs = 3
n_classes = 10
batch_size = 128
chunk_size = 28
n_chunks = 28
rnn_size = 128

# Defining some placeholders in our graph
x = tf.placeholder('float', [None, n_chunks, chunk_size])
y = tf.placeholder('float')


def rnn_model(data):
    # Initialise the weights and biases of the rnn layer randomly
    layer = {'weights': tf.Variable(tf.random_normal([rnn_size, n_classes])),
             'biases': tf.Variable(tf.random_normal([n_classes]))}

    # Fit the data to the TensorFlow requirements
    data = tf.transpose(data, [1, 0, 2])
    data = tf.reshape(data, [-1, chunk_size])
    data = tf.split(0, n_chunks, data)

    # Create the LSTM cell
    lstm_cell = rnn_cell.BasicLSTMCell(rnn_size, state_is_tuple=True)
    # Create the rnn
    outputs, states = rnn.rnn(lstm_cell, data, dtype=tf.float32)

    # Calculate the output
    output = tf.matmul(outputs[-1], layer['weights']) + layer['biases']

    return output
```

```
def train_neural_network(x):
    # Produces the initial predictions
    prediction = rnn_model(x)
    # Measure how wrong our predictions are
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(prediction, y))
    # We use AdamOptimizer as our cost optimisation function
    optimizer = tf.train.AdamOptimizer().minimize(cost)

    # Start a session
    with tf.Session() as sess:
        # Initialise all variables
        sess.run(tf.global_variables_initializer())

        # Train for the number of epochs
        for epoch in range(hm_epochs):
            # We will keep track of our error
            epoch_loss = 0
            # Train for every batch
            for _ in range(int(mnist.train.num_examples/batch_size)):
                # get the x and y data for this epoch (from the MNIST data set)
                epoch_x, epoch_y = mnist.train.next_batch(batch_size)
                epoch_x = epoch_x.reshape((batch_size, n_chunks, chunk_size))

                # Run the optimiser and cost function on our data
                _, c = sess.run([optimizer, cost], feed_dict={x: epoch_x, y: epoch_y})
                epoch_loss += c
            # Print our progress
            print('Epoch', epoch, 'completed out of', hm_epochs, 'loss', epoch_loss)

        # Count the number of correct predictions
        correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(y, 1))
        # Check our prediction accuracy on the test data set
        accuracy = tf.reduce_mean(tf.cast(correct, 'float'))
        print('Accuracy:', accuracy.eval({x: mnist.test.images.reshape((-1, n_chunks,
                                          y: mnist.test.labels}))

# Run the training method
train_neural_network(x)
```

Appendix Q

Number Recognition using Neural Nets

In this appendix you find the implementation of a neural network using TensorFlow to implement a number recognition program.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

# In this file we use the MNIST data set and TensorFlow to predict the number of a giv
# written number from 0 to 9.

# Our data set
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# Start building our model by defining the number of nodes in every hidden layer
n_nodes_hl1 = 500
n_nodes_hl2 = 500
n_nodes_hl3 = 500
# The number of classes, 0 to 9 so 10 classes
n_classes = 10
# Batch size to train the network
batch_size = 100

# Defining some placeholders in our graph, 784 because we are working on 28 by 28 pix
# the shape of the variable for TensorFlow)
x = tf.placeholder('float', [None, 784])
y = tf.placeholder('float')


def neural_network_model(data):
    # Defines the weights and biases for the network randomly
    # biases here are a value that is added to our sums (so that our network still wo
    # a 0), these biases will need to be op
    hidden_1_layer = {'weights': tf.Variable(tf.random_normal([784, n_nodes_hl1])),
                      'biases': tf.Variable(tf.random_normal([n_nodes_hl1]))}
    hidden_2_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl1, n_nodes_h
                      'biases': tf.Variable(tf.random_normal([n_nodes_hl2]))}
    hidden_3_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl2, n_nodes_h
                      'biases': tf.Variable(tf.random_normal([n_nodes_hl3]))}
    output_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl3, n_classes]))
                    'biases': tf.Variable(tf.random_normal([n_classes]))}

    # Here we start the feed forward flow
    # layer 1 = input data * weights + biases
```

```
    l1 = tf.add(tf.matmul(data, hidden_1_layer['weights']), hidden_1_layer['biases'])
    l1 = tf.nn.relu(l1)
    # layer 2 = layer 1 * weights + biases
    l2 = tf.add(tf.matmul(l1, hidden_2_layer['weights']), hidden_2_layer['biases'])
    l2 = tf.nn.relu(l2)
    # layer 3 = layer 2 * weights + biases
    l3 = tf.add(tf.matmul(l2, hidden_3_layer['weights']), hidden_3_layer['biases'])
    l3 = tf.nn.relu(l3)
    # output = layer 3 * weights + biases
    output = tf.matmul(l3, output_layer['weights']) + output_layer['biases']

    return output


def train_neural_network(x):
    # Produces the initial predictions
    prediction = neural_network_model(x)
    # Measure how wrong our predictions are
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(prediction, y))
    # We use AdamOptimizer as our cost optimisation function
    optimizer = tf.train.AdamOptimizer().minimize(cost)

    # Define the number of epochs
    hm_epochs = 10
    # Start a session
    with tf.Session() as sess:
        # Initialise all variables
        sess.run(tf.global_variables_initializer())

        # Train for the number of epochs
        for epoch in range(hm_epochs):
            # We will keep track of our error
            epoch_loss = 0
            # Train for every batch
            for _ in range(int(mnist.train.num_examples/batch_size)):
                # get the x and y data for this epoch (from the MNIST data set)
                epoch_x, epoch_y = mnist.train.next_batch(batch_size)
                # Run the optimiser and cost function on our data
                _, c = sess.run([optimizer, cost], feed_dict={x: epoch_x, y: epoch_y})
                epoch_loss += c
            # Print our progress
            print('Epoch', epoch, 'completed out of', hm_epochs, 'loss', epoch_loss)

        # Count the number of correct predictions
        correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(y, 1))
        # Check our prediction accuracy on the test data set
        accuracy = tf.reduce_mean(tf.cast(correct, 'float'))
        print('Accuracy:', accuracy.eval({x: mnist.test.images, y: mnist.test.labels}))

# Run the training method
train_neural_network(x)
```

Appendix R

Sentiment Analyses with Neural Nets

In this appendix you find the implementation of a sentiment analyses using Neural Networks from TensorFlow.

```
import os
import tensorflow as tf
import numpy as np
from NeuralNetwork.create_sentiment_featuresets import create_feature_sets_and_labels

# In this file we will use TensorFlow to correctly identify the sentiment of a given

# Our root directory of this file (In the Neural Network folder)
ROOT_DIR = os.path.dirname(os.path.abspath(__file__))

# Our data set
train_x, train_y, test_x, test_y = create_feature_sets_and_labels(ROOT_DIR + '/../data
                                                                  ROOT_DIR + '/../data

# Start building our model by defining the number of nodes in every hidden layer
n_nodes_hl1 = 1500
n_nodes_hl2 = 1500
n_nodes_hl3 = 1500
# The number of classes, 0 to 9 so 10 classes
n_classes = 2
# Batch size to train the network
batch_size = 100

# Defining some placeholders in our graph, 784 because we are working on 28 by 28 pix
# the shape of the variable for TensorFlow)
x = tf.placeholder('float')
y = tf.placeholder('float')


def neural_network_model(data):
    # Defines the weights and biases for the network randomly
    # biases here are a value that is added to our sums (so that our network still wo
    # a 0), these biases will need to be op
    hidden_1_layer = {'weights': tf.Variable(tf.random_normal([len(train_x[0]), n_nod
                      'biases': tf.Variable(tf.random_normal([n_nodes_hl1]))}
    hidden_2_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl1, n_nodes_h
                      'biases': tf.Variable(tf.random_normal([n_nodes_hl2]))}
    hidden_3_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl2, n_nodes_h
                      'biases': tf.Variable(tf.random_normal([n_nodes_hl3]))}
    output_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl3, n_classes]))
                    'biases': tf.Variable(tf.random_normal([n_classes]))}
```

```
    # Here we start the feed forward flow
    # layer 1 = input data * weights + biases
    l1 = tf.add(tf.matmul(data, hidden_1_layer['weights']), hidden_1_layer['biases'])
    l1 = tf.nn.relu(l1)
    # layer 2 = layer 1 * weights + biases
    l2 = tf.add(tf.matmul(l1, hidden_2_layer['weights']), hidden_2_layer['biases'])
    l2 = tf.nn.relu(l2)
    # layer 3 = layer 2 * weights + biases
    l3 = tf.add(tf.matmul(l2, hidden_3_layer['weights']), hidden_3_layer['biases'])
    l3 = tf.nn.relu(l3)
    # output = layer 3 * weights + biases
    output = tf.matmul(l3, output_layer['weights']) + output_layer['biases']

    return output


def train_neural_network(x):
    # Produces the initial predictions
    prediction = neural_network_model(x)
    # Measure how wrong our predictions are
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(prediction, y))
    # We use AdamOptimizer as our cost optimisation function
    optimizer = tf.train.AdamOptimizer().minimize(cost)

    # Define the number of epochs
    hm_epochs = 10
    # Start a session
    with tf.Session() as sess:
        # Initialise all variables
        sess.run(tf.global_variables_initializer())

        # Train for the number of epochs
        for epoch in range(hm_epochs):
            # We will keep track of our error
            epoch_loss = 0
            # Train for every batch, we manually select the batch
            i = 0
            while i < len(train_x):
                # Start point of this batch
                start = i
                # End point of this batch
                end = start + batch_size
                # Select the batch for the x set and y set
                batch_x = np.array(train_x[start:end])
                batch_y = np.array(train_y[start:end])
                # Run the optimiser and cost function on our data
                _, c = sess.run([optimizer, cost], feed_dict={x: batch_x, y: batch_y})
                epoch_loss += c
                i += batch_size
            # Print our progress
            print('Epoch', epoch, 'completed out of', hm_epochs, 'loss', epoch_loss)

        # Count the number of correct predictions
        correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(y, 1))
        # Check our prediction accuracy on the test data set
        accuracy = tf.reduce_mean(tf.cast(correct, 'float'))
        print('Accuracy:', accuracy.eval({x: test_x, y: test_y}))
```

```
# Run the training method
train_neural_network(x)
```

Appendix S

Convolution with TFLearn

In this appendix you find the implementation of a convolutional neural net using TFLearn.

```
import tflearn
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.core import input_data, dropout, fully_connected
from tflearn.layers.estimator import regression
import tflearn.datasets.mnist as mnist

# This file shows an abstraction of the convolution implementation used in tensorflow-

# Load the mnist data
X, Y, test_x, test_y = mnist.load_data(one_hot=True)

# Reshaping
X = X.reshape([-1, 28, 28, 1])
test_x = test_x.reshape([-1, 28, 28, 1])

# Building the CNN
# Input layer
convnet = input_data(shape=[None, 28, 28, 1], name='input')

# 2 layers of convolution and pooling
convnet = conv_2d(convnet, 32, 2, activation='relu')
convnet = max_pool_2d(convnet, 2)
convnet = conv_2d(convnet, 64, 2, activation='relu')
convnet = max_pool_2d(convnet, 2)

# Fully connected layer, with dropout
convnet = fully_connected(convnet, 1024, activation='relu')
convnet = dropout(convnet, 0.8)

# Output layer
convnet = fully_connected(convnet, 10, activation='softmax')
convnet = regression(convnet, optimizer='adam', learning_rate=0.01, loss='categorical_

# Create the model
model = tflearn.DNN(convnet)

# Train the model
model.fit({'input': X}, {'targets': Y}, n_epoch=10, validation_set=({'input': test_x}
          snapshot_step=500, show_metric=True, run_id='mnist')

# Use model to predict
# model.predict(x)
```

```
# Save model
# model.save(filename)

# Load model,, you'll still need to set the structure of the model though, since load
# model.load(filename)
```