Automated collision detection using machine learning algorithms on sensor data

Project Report

from the Course of Studies Angewandte Informatik

at the Cooperative State University Baden-Württemberg Mannheim

by

Tim Schmidt

August 2018

Time of Project	17 weeks
Student ID, Course	8531806, TINF15AI-BC
Company	IBM Deutschland GmbH, Ehningen
Supervisor in the Company	Julian Jung
Reviewer	Julian Jung

Author's declaration

Hereby I solemnly declare:

1. that this Project Report, titled *Automated collision detection using machine learning algorithms on sensor data* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. I have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Project Report has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. I have not published this Project Report in the past;
5. the printed version is equivalent to the submitted electronic one.

I am aware that a dishonest declaration will entail legal consequences.

Mannheim, August 2018

Tim Schmidt

Contents

List of Figures

1 Introduction

1.1 Motivation

The field of data analytics and machine learning is increasingly becoming the main differentiator of industry competition. This is especially true in combination with another growing field, the Internet of Things (IoT). This is what the McKinsey Global Institute's study 'The Age of Analytics: Competing In A Data-Driven World' found. [1] IBM has set a goal to tackle the challenges of IoT and Big Data with an array of new services, industry offerings and capabilities for enterprise clients, startups and developers. [2] These services are part of IBM Bluemix, a cloud plaform as a service (PaaS).

An important step for IBM is getting clients interested and aware of the capabilities of IoT and machine learning so that collaborations and projects with clients can be established. One way this is done is by presenting showcases – small projects that show the possibilities and values of the technology in an exemplary way. It is essential to show technologies from the IBM Bluemix portfolio to present their capabilities and outline differentiating factors to competitors.

The Sensorboard is such a showcase. A physical skateboard is equipped with a Texas Instrument's SensorTag and able to connect wirelessly to the IBM's Bluemix cloud platform. From there one or more Sensorboards can be monitored and managed. As part of the showcase a the Sensorboard is demonstrated as a rentable and cloud-managed vehicle for urban mobility.

This showcase will be extended with machine learning applications to present the value machine learning algorithms can have on sensor data. The specific goal of that extension is to recognize collisions that happen to the Sensorboard and that indicate accidents. This recognition will be based solely on sensor data and patterns in the time series data. For this IBM Bluemix services are used to showcase the capabilities of the platform. There are services which are requested to be demonstrated and used in the showcase. This is the IBM IoT Foundation, used to connect the Sensorboard to the platform. Moreover the IBM Data Science Experience should be used as a development environment for an underlying Apache Spark cluster, on which machine learning models will can be created.

1.2 Objective

A consumer grade skateboard is equipped with a Texas Instrument's SensorTag. The Texas Instrument's SensorTag collects acceleration data from the skateboard on three spacial axis. To process this data cloud services of IBM Bluemix are used. This especially includes the IBM Data Science Experience and an Apache Spark service. Consequently utilizing Apache Spark as a framework for cluster computing, machine learning models are trained using Apache Spark. This narrows down the range of machine learning methods to the ones available for processing on Apache Spark. Such methods are provided by the Apache Spark MLlib function library.

This paper investigates on the feasibility of machine learning algorithms for collision detection in the above described setting and identifies, if feasible, the best performing algorithm under the described conditions.

The scenario of impact detection that may indicate accidents requires a higher focus on detecting an accident than on preventing false alarms. A metric is found to represents this unequal relationship in the assessment of algorithm performance.

1.3 Limitations of the Research

The technologies described - the Texas Instrument's SensorTag, the IBM Bluemix services and function libraries - are a given for the project by IBM. Therefore the decisions for those and no other technologies will not be discussed and no evaluation of alternatives will be conducted in this paper.

The findings will only be acquired and applicable for the specific hardware and configuration described in the paper. Findings can not be transferred to other vehicles, sensors, configurations or any other setup than the one described.

The data which is used to come to decisions and to train models is generated in experiments, which are conducted solely for this reason. No other sensor data of skateboards or other vehicles are used in the considerations of this paper. The machine learning algorithms examined only include algorithms currently provided by the function library Apache Spark MLlib.

1.4 Organization of the Research

The chapter 'Theory' introduces the topic of machine learning. Relevant machine learning theory is explained with sufficient theoretical and mathematical background information.

The goal of the chapter is to create a basic understanding for methods and metrics this paper investigates.
In the following chapter 'Analysis of the pattern recognition problem', the pattern recognition problem will be analysed in detail. The goal of the chapter is to define specific possible solutions for the identified problem.
The chapter 'Algorithm configuration' brings together the theoretical knowledge about machine learning and possible solution strategies. It is shown how algorithms are best configured and applied for the problem. The goal of the chapter is to generate a comparable result for the different possible solutions.
In the chapter 'Evaluation', the results are then compared to each other. The goal of this chapter is to answer the question of whether machine learning algorithms are feasible for collision detection. If feasible, a winning solution that is best performing under the considered conditions is identified. A conclusions about the results is drawn.
In the end, the chapter 'Reflection and outlook' presents a critical reflection of the results and a proposal on possible further research and development.

2 Theory

2.1 Machine Learning

Historically computers had to be programmed on a specific task using domain knowledge of humans. Machine learning challenges this traditional way of programming. Rather than trough extensive domain knowledge, machine learning algorithms are used to autonomously learn from data and information.

This intention is often represented by a 'learning' f*input-output function.* Assume there exists function, f, which represents a true behaviour or correlation. The task of any machine learning algorithm is thereby to find a *hypothesis function* h that resembles the behaviour of f as closely as possible. A function h is also called *model.* Both f and h are function of a fixed length vector-valued input $X = x_1, x_2, ..., x_n$ which has n components. Those might be n attributes of an observed object. An input x is often referred to as a *data point.* The components of a data point x are the *features* of x. The features of a data point span the *feature space.* The feature space refers to the n-dimensional space in which the features of a data point are represented.
The function h can be thought of as a device that has X as an input and $h(X)$ as an output. The hypothesis function, h, is selected based on a *trainings set*, Ξ, of m input vector examples. The process of selecting a function h, based on a *trainings set* Ξ, is called *training* a model or also *fitting* a model.
There are two major settings in machine learning.
The first setting is *unsupervised learning.* In this case the trainings set of vectors Ξ doesn't provide information about the value $f(X)$. The goal of the unsupervised learning algorithms usually is to partition the training set into subset, Ξ $_1$, Ξ $_2$, ..., Ξ $_\mathrm{r}$. An example for unsupervised machine learning learning is *clustering*, which finds clusters and discovers relationships in the data.
The second is *supervised learning*, in which the values of f for the samples in the training set Ξ are known. The value of f corresponding to a data point x is also called the *label* of x. A dataset with a label for every data point in it is referred to as a *labelled* dataset. The assumption is that a hypothesis h is a good guess for f when $h(x_i)$ agrees with the values of $f(x_i)$ or the given labels for the majority of samples x_i .
In supervised machine learning two main areas are distinguished, *Regression* and *Classifi-*

cation.
In Regression the hypothesis h has a continuous output. Regression could, for instance, be used to approximate a function for a stock price and predict a future stock price as a numerical value on a continuous range of possible prices.
In Classification the hypothesis h has a categorical output. Classification could, for instance, be used to classify emails to either spam or no spam. It also finds application in patter recognition. The case of an classification task that has a need to classify data into two distinguished classes is called *binary classification.* The alternative case is *multiclass classification*, which aims to classify data points into one of multiple different classes.

2.2 Evaluation of Classifiers

Assessing the quality of a model is essential to choosing the right method or configuration of an algorithm. This is why in this section, different metrics are introduced to empirically measure the quality of a *binary classifiers.* Also a way of consistently checking for indicating metrics with an *validation scheme* is shown.

Evaluation metrics

In the following a binary classification task is regarded. The data set used has samples (data points) of the classes 1 and 0. A model $h(X)$ predicts a class label for for a given sample X. Assume the samples of the class 1 are relevant and the model is examined in regards to the class 1.
The *positives*(pos) are the set of all samples where the class label is 1. In contrast the emphnegatives (neg) are the set of all samples where the class label is not 1 (in the binary setting 0).
The model $h(X)$ also predicts some samples as positives and some as negatives. The prediction doesn't always coincide with the true class labels. The samples that the model identifies as positives and that are actually labelled as positives are called *true positives* (tp). In contrast the samples that the model identifies as positives but that are labelled as negatives are called *false positives* (fp). Similarly the samples that the model identifies as negatives and that are actually labelled as negatives are called *true negatives* (tn). Again, in contrast the samples that the model identifies as negatives but that are labelled as positives are called *false negatives* (fn).

One of the most basic metrics of a classifier are it's *precision* and *recall.*
Precision is defined as the ratio of true positives to all positives the model h identified (true positives + false positives):

$$\text{Precision} = \frac{tp}{tp + fn} \tag{2.1}$$

The precision can be interpreted as a measure of how good a model is at identifying only the positives as positive while avoiding to misclassify negatives as positives.
Recall is defined as the ratio of true positives to all positives:

$$\text{Precision} = \frac{tp}{pos} \tag{2.2}$$

The recall can be interpreted as a measure of how good a model is at identifying all the positives and missing as little positives as possible.

Intuitively it can be seen that precision and recall have a inverse relationship to each other. For instance, the more samples are identified as positives by a model, the higher is the true positive rate, hence the recall. But the chance of identifying negatives as positives also rises such that the precision decreases.
The goal of creating a model with any given method is to increase both, precision and recall, simultaneously. Nevertheless in some use cases it may be of advantage to value a good recall higher than a good precision and vice versa. This might be the case if it is very important to find all the positives, while not caring that much about false negatives ('false alarms'), when negatives are incorrectly identified as positives.

A way to merge the two metrics in a joined metric, both for the case of equal and unequal importance, is the *f-score*, also called f_β*-score.*
The f$_\beta$-score takes into account both recall and precision and can be configured to weight one more important than the other. The f$_\beta$-score is defined as follows:

$$f_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} \tag{2.3}$$

In equation 2.3 the parameter β is used to put more weight on precision or recall. If $\beta > 1$ if puts more weight on the recall, if $\beta < 1$ more weight is put on the precision
For instance a f$_2$-score would put more weight on the recall of a classifier:

$$f_\beta = (1 + 2^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(2^2 \cdot \text{precision}) + \text{recall}} \tag{2.4}$$

On the other side a $f_{0.5}$-score would put more weight on the precision of a classifier:

$$f_\beta = (1 + 0.5^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(0.5^2 \cdot \text{precision}) + \text{recall}} \tag{2.5}$$

In the case of equal importance, the f_1-score is used, which is the *harmonic mean* of precision and recall:

$$f_\beta = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{2.6}$$

A weighted or not weighted f_β-score can be used to compare different approaches to a classification problem to each other.

Validation scheme

To assess the recall, precision or other performance metrics of a model it has to be tested. Testing a model is done with data, that the model hasn't 'seen' yet, meaning data points that weren't in the training dataset. This is usually done by selecting a small fraction of all the available data and setting it aside as *testing data*. This data is used to test a model and evaluate it's performance later on. The remaining data is used to train or fit the model. It is called *training data*.
Doing this random split into training and testing data can lead to problems: Depending on the split that is done the testing data may by chance be very hard to classify in comparison to the training data. This would suggest a bad performance of the classifier later on. On the other side, if the training data is easy to classify in comparison to the training data later evaluation would show a better performance than it might is in reality.
To counteract this phenomenon *cross-validation* is done. Cross-validation involves partitioning a the available data into training and testing data multiple times, evaluation for a specific metric, and average the results. A cross-validation approach that is relevant for this paper is *k-fold validation*.

K-fold validation is a non-exhaustive cross validation method, meaning it does not compute all the possible ways of spitting the available data into subsections.
In k-fold validation the available data is split into k equal sized, non-overlapping subsets of data points. In each validation step one of the subsets is seen as the testing data and the rest is combined to act as training data. A model is now trained on the training data and tested for different metrics, like recall and precision, on the training data. This process is

k times, each time leafing one subset (*fold*) out of training as a testing set. The metrics are then averaged.

Cross-validation averages many tests with different test and training data to achieve a consistent evaluation of a classifier with low variance between test.

2.3 Algorithms

This section will introduce basic concepts and specific algorithms for the classification setting. The purpose of this section is to give a introduction into methods and tools used in this paper and give the reader a understanding of the underlying principles. Some topics may be oversimplified or not discussed in their whole complexity. The reason for this is that it is beyond the scope of this theory section, which is giving a introduction to machine learning methods relevant for this paper.

Before getting into specific algorithms, a few basic concepts, relevant for most classification algorithms, are discussed.

Hyper-parameters:
Hyper-parameters are parameters that configure a machine learning algorithm itself. Changing the parameters of a algorithm will the the way the algorithm constructs the model h. An example for this would be the maximal height of a decision tree in a decision tree model (see section 'Decision tree'). The task of finding good hyper-parameter for a algorithm is called *hyper-parameter tuning* or *hyper-parameter optimization*.

Model linearity:
A classification model classifies data points according to their feature vector. Visualized in a n-dimensional feature space for a feature vector of length n this means classification depends on the position of the data point in the feature space. The decision of a model to classify a region in feature space as one class and not another can be represented by a *decision boundary*. Data points laying on the one side of the decision boundary will be predicted by h to belong to one class while data points on the other side belong to an other class.
This decision boundary of a model can have any form. If the form of the decision boundary is linear, meaning the decision boundary can be represented by a linear function of the form

$$f(x) = \beta_0 + \beta_1 X_1 + ... + \beta_n X_n \tag{2.7}$$

in a n-dimensional feature space, the model is *linear*.
If this isn't the case the model is *non-linear*.

Bias-variance tradeoff:
Assume a model tries to approximate a function f.

The *bias error* of a model are errors like misclassification due to simplified assumptions made by a model. The bias of a model are these simplifying assumptions.
Low bias of a model means less assumptions about a function f are made.
High bias of a model means more assumptions about a function f are made.

The *variance error* of a model are errors like misclassification due to simplified assumptions made by a model. The bias of a model are these simplifying assumptions.
Low variance of a model means changes in the training data, like added noise, will not have a big impact on the resulting model.
High variance of a model means changes in the training data, like added noise, will have a big impact on the resulting model.

Bias and variance have a inverse relationship to each other. Low Bias and high variance leads to *overfitting*. This means a model makes little assumptions about the form of f and rather 'memorizes' the training data points. The model doesn't *gerneralize* very well.

High Bias and low variance leads to *underfitting*. This means a model makes too many assumptions about the form of f misses a lot of more subtle relationships in the data. The model *gerneralize* the problem to much.

Loss function and regularization:
loss function / regularization

In the following sections the basics of certain relevant classification algorithms will be explained.

2.3.1 Linear support vector classifier

In this chapter the support vector classifier will be introduced. The method was developed in the 1990's and is often used as one of the best performing 'out of the box' classifiers. The goal of a linear support classifier is to find a function for a *hyperplane* that splits the data points of two classes as best as possible. The support vector classifier is a extension of a maximal margin classifier. First the concept of classification using a hyperplane and with a maximal margin classifier will be explained. Then the extension to a support vector classifier will be shown. A model based on a support vector classifier is linear.

Hyperplane classification

The goal of the maximal margin classifier is to classify data by finding a hyperplane that optimally separates two classes in a dataset. Data points are either in class $y = f(x) = -1)$ or $y = f(x) = 1)$.
A hyperplane is a flat affine subspace of dimension $p - 1$, where p is the number of dimensions the hyperplane is in. In the case of a maximal margin classifier the number of dimensions which a hyperplane will be constructed in is the number dimension of the feature space. This stems from the intention to separate data points which have a feature vector of p dimensions. Accordingly a hyperplane is defined by

$$\beta_0 + \beta_1 X_1 + ... + \beta_p X_p = 0 \tag{2.8}$$

for a feature space of dimension p. It can be said that β_0 to β_p 'define' the hyperplane. If a point $X = (x_1, x_2, ..., x_p)$ lies on the hyperplane, the point features x_1 to x_p would satisfy the equation 2.7.
If this doesn't hold true but

$$\beta_0 + \beta_1 X_1 + ... + \beta_p X_p > 0 \tag{2.9}$$

is true, the point X lies on the one side of the plane, whereas if

$$\beta_0 + \beta_1 X_1 + ... + \beta_p X_p < 0 \tag{2.10}$$

is true, the the point X lies on the other side.
If a hyperplane is defined in a way that the data points of on class lay on the one side of the hyperplane and the data points of the other class on the other side, any given data point can be classified accordingly as either one or the other class according to equations 2.8 and 2.9.
The *magnitude* of $h(X)$ can be used to derive a confidence score for the classification. A higher magnitude of $h(X)$, which can be interpreted as distance to the plane, corresponds to a higher confidence in the classification.

Maximal margin classifier

If a class separating hyperplane exist there exists an infinite number of such hyperplanes separating the classes (a hyperplane can always be moved or varied slightly while still remaining class separating). The question arises which hyperplane should be used.
A maximal margin classifier uses the hyperplane, where the distance to the next data point is the biggest under all possibilities. The distance to the next data point is called the *margin*. A hyperplane that has the biggest possible margin is a *maximal margin hyperplane*. Data points where the distance to the maximal margin hyperplane is equal to the margin are the *support vectors* as seen in figure 2.1. The maximal margin hyperplane depends on the support vectors because moving or removing the support vectors would change the hyperplane. Moreover the vectors that are further away from the hyperplane than the support vectors don't contribute to definition of the hyperplane. This allows the hyperplane to be soly defined by the support vectors.

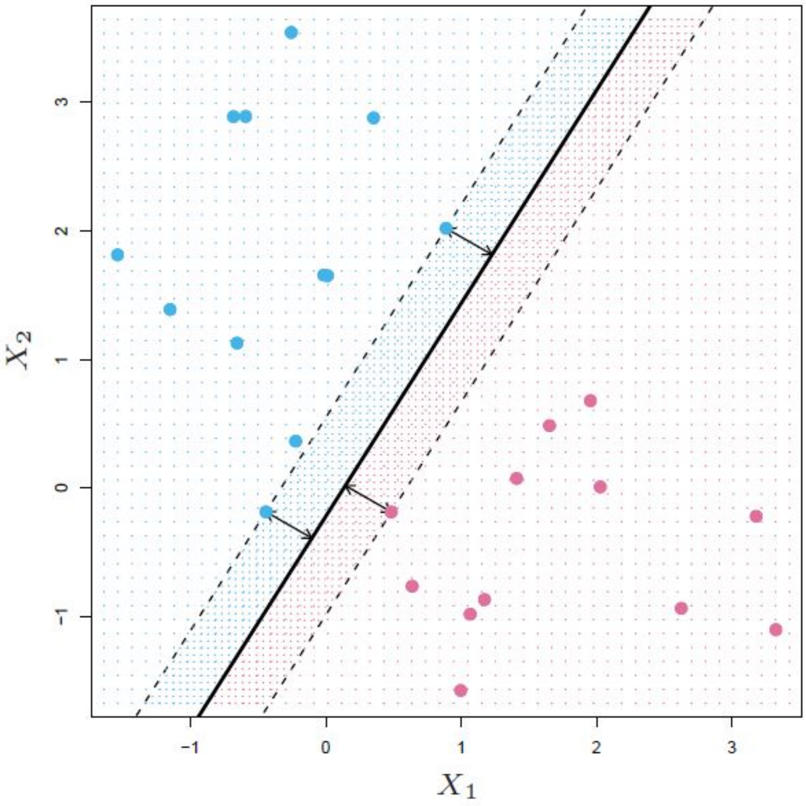


Figure 2.1: Maximal margin hyperplane separating two classes of data points with the margin borders visualized by dotted lines

To find a maximal margin hyperplane the parameters of the hyperplane have to be set in a way that maximises the size of the margin M and

$$y_i(\beta_0 + beta_1 x_i 1 + \beta_2 x_i 2 + ... + \beta_p x_i p) \leq M \forall i = 1, ..., n \tag{2.11}$$

holds. y_i is the true class of the ith data point. If all data points are classified correctly, which is the goal, the left side of the equation 2.10 is always positive. This is because either both y_i and $f(X_i)$ are negative or both are positive. The right side of the equation makes sure that every data point has a minimal distance of M to the hyperplane.

The maximal margin classifier has some faults that have to be considered. Because the data points are of a class are strictly separated by a hyperplane of a linear form from the other class, it is not possible to find a hyperplane if the classes are not linear separable. Furthermore even when the classes are linear separable the hyperplane pound might be forced in a unfavourable spot by only one single data point that doesn't represent most of the other data points of the same class. The extension to a support vector classifier solves those problems.

Support vector classifier

To solve the above mentioned faults the *support vector classifier*, also called *soft margin classifier*, allows some data points of the training data to be not on the correct side of the margin or the hyperplane.
To find a hyperplane for a support vector classifier the parameters of the hyperplane have to be set in a way that maximises the size of the margin M and the equations

$$y_i(\beta_0 + beta_1 x_i 1 + \beta_2 x_i 2 + ... + \beta_p x_i p) \leq M(1 - \epsilon_i) \forall i = 1, ..., n \tag{2.12}$$

$$\epsilon_i >= 0, \sum_i = 1^n \epsilon_i \leq C \tag{2.13}$$

both hold.
The concept is similar to the one seen in equation 2.10. Differentiating here are the *slack variables* $\epsilon_1, ..., \epsilon_n$. ϵ_i indicates where the ith data point is located relative to the hyperplane and it's margin. If the ith data point is on the correct side of the margin $\epsilon_i = 0$. If the ith data point is on the correct side of the hyperplane but on the wrong side of the margin it *violates* the margin and $\epsilon_i = 0$. If the ith data point is on the wrong side of the hyperplane then $\epsilon_i = 0$.
C is a tuning parameter that limits the total sum of e_is.
A bigger C allows more datapoints to violate the margin or be on the wrong side of the hyperplane.
This way a hyperplane can be found even when the classes are not strictly linearly separable and isolated outlier data points that would distort the resulting hyperplane can be mostly

ignored.

With a support vector classifier it's now possible to create a model h that can be trained on data with non linear separable classes. The model takes a data point and classifies it to either one class ($h(x) < 0$) or the other class ($h(x) > 0$). The magnitude $|h(x)|$ can be interpreted as confidence score of the classification.

2.3.2 Logistic regression

In this chapter the logistic regression will be introduced. The goal of logistic regression is to create a classifier which predicts the probability for a given data point to belong to a certain class. A model based on a support vector classifier is linear.

The probability of a data point X with p parameters belonging to a class can be written as the probability of a class c under the condition of X:

$$p(c|X) \in [0,1] \tag{2.14}$$

The model $h(X) = p(c = 1|X)$ gives the probability of a data point X being of the class 1.

Logistic regression is based on a linear regression. Linear regression models a linear function. It tries to choose the function parameters $\beta_0, ..., \beta_p$ in a way that the function approximately goes through every data point in the training data it was derived from. 2.14.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + +\beta_n x_n \tag{2.15}$$

How this can be done exactly is out scope for this paper. The focus is on the classification algorithm logistic regression.
Logistic regression tries to create a model predicting probability, hence a value $p \in [0,1]$. The *standard logistic function*, as seen in 2.15, has the property of mapping an input to a range of 0 to 1. The codomain of the standart logistic function is 0 to 1.

$$f(x) = \frac{1}{1 + e^{-}x} = \frac{e^x}{1 + e^x} \tag{2.16}$$

Utilizing the linear regression in equation 2.14 the output is now transformed with the standart logistic function:

$$h(x) = \frac{e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_n x_n}}{1+e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots++\beta_n x_n}} \tag{2.17}$$

The function parameters β_1 to β_n are estimated based on the given data points.
This way logistic regression creates a model which predicts the probability of a data point being in a certain class.

D. R. Cox, The Regression Analysis of Binary Sequences 1958

2.3.3 Decision tree

In this chapter the decision tree will be introduced. The goal of a decision tree is to create a classifier which splits the segments the feature space into subspaces and classifies new data point according to which subspace it is in. A model based on a decision tree is non-linear.

The task that a decision tree accomplishes is splitting the feature space into meaningful slices with which the create model can classify new data points. This is done by splitting the feature space multiple times recursively. That means After a split two created subspaces are then further split. This recursive splitting can also be visualized in a binary tree. This binary tree is called a *decision tree*. Each node represents a split of the feature (sub-)space. Only the terminal nodes don't represent a split and rather an area of the feature space that corresponds to a specific class. First the decision making process is explained. Then the mapping of a terminal node to a class is explained.
The tree is constructed iteratively and top-down (starting at the root of the tree). This is often called *recursive binary splitting*. Construction follows a *greedy* approach, meaning a node is created only considering the benefit that comes with a certain splitting rule for this split, not looking into the future and assessing the benefit of a certain splitting rule for the later tree.
Splitting a space of any dimension into two subspaces needs two aspects: The axis on which the split is done and the place on the axis where the split is done. Accordingly a splitting rule can be written as follows:

$$a \leq c \tag{2.18}$$

where a represents one of p axis in an p-dimensional feature space and c represents a value in the domain of axis a. This decision rule splits the feature space orthogonal to the axis a where the value$of a = c$.
The question arises how this decision rule should be chosen for any given feature (sub-)space. In the ideal case the a split is placed in a way that separates most data point from one class from the data points of the other class and leaves two most 'pure' subspaces. The *impurity* of a subspace is most commonly measured by the following metrics.
Gini impurity:

$$\sum_{i=1}^{C} f_{\mathrm{i}}(1 - f_{\mathrm{i}}) \tag{2.19}$$

C is the number of unique labels and f_{i} is the frequency of label i in a created feature subspace. For binary classification C is maximal 2.
Entropy:

$$\frac{1}{N} \sum_{i=1}^{C} -f_{\mathrm{i}} \log(f_{\mathrm{i}}) \tag{2.20}$$

C is the number of unique labels and f_{i} is the frequency of label i in a created feature subspace. For binary classification C is maximal 2.

To reach the highest 'purity', either one of the metrics has to be minimized.
Finding the optimal split requires finding a and c such that the splitting rule $a \leq c$ minimizes the sum of the impurity measures for both forming feature subspaces. The impurity measures commonly used are Ginni impurity (equation 2.18) and Entropy (equation 2.18). The iterative splitting or growing of the tree is stopped when either a certain height of the tree is reached, a minimal number of trainings data points in one terminal node is undercut or the purity of a subspace can't be significantly improved by another split.

Now that a decision tree is constructed every terminal node is assigned a class prediction. The class prediction of a terminal node is simply the most common class of the data points in the feature subspace that the terminal node represents. For later ensemble methods based on a decision tree it is important to note, that a terminal node t can not only predict but also give the probability $p(C_i|X)$ of a class C_i:

$$p_t(C_i|X) = \frac{\text{number of data labeles as class } C_i \text{ in terminal node } t}{\text{total number of data points in termin node } t} \tag{2.21}$$

A decision tree model can classify any new data point by passing it from the root of the decision tree to a terminal node, which holds the class prediction, by simply following the decision rules either to the right to the left child node (depending on how the tree is implemented) if the decision rule (as seen in equation **??**) holds and to the other side if it doesn't.

Decision trees have the advantage that the decision rules of a model are easy to understand and can be displayed graphically. But a decision tree has also disadvantages, especially a high variance is often occurring, which leads to overfitting. This problem is counteracted by using *decision tree assembles* like *random forest* or *boosted trees*. The two algorithms are explained in the following.

2.3.4 Random forest

In this chapter random forests will be introduced. The method utilizes decision trees and creates an assemble of trees improving, among others, on the high variance of a single decision tree. A model based on a random forest is non-linear.

To reduce variance the method uses the fact that averaging a set of observations reduces variance in the resulting observation. *Bagging* (also called *bootstrap aggregation* utilizes exactly that fact. From the singe training set Ξ, B subset are being created. On every subset of Ξ a decision tree is constructed. The construction of a tree on a single bootstrapped subset is described below. The different models will then all vote in a *majority vote* to come to a conclusion.

Additional to bagging the random forest has another speciality when training models. Only bagging decision trees would lead to very similar trees. Random forest prevents this during the tree construction phase. A decision tree algorithm decides on a split, in particular on the axis or feature, and on the value where the split is done. A random forest algorithm prohibits the decision for some axis, each time a decision rule is created. In other words, for every decision rule there is a randomly chosen subset of all axis or features possible to choose from.

This way the construction algorithm is encouraged to make splits that are not always optimal but may lead to a better performing decision the in the long run by chance.

To make predictions on a new data point a majority vote across all decision trees is conducted and the winning class is predicted for the data point. The method reduces variance and minders the risk of overfitting. Another method that improves prediction rules is boosting and will be discussed in the next chapter.

2.3.5 Boosted trees

In this chapter Boosted trees will be introduced. The method also utilizes decision trees and tackles the problem of the high variance of a single decision tree. Similar to a random forest, boosted trees use a ensemble of decision trees. Boosting involves constructing multiple decision trees successively with every new decision tree encouraged to counteract the shortcomings of the previous constructed trees. This is done by fitting each decision tree on a modified version of the training set of the previous decision tree. A model based on boosted trees is non-linear.

The way a boosted tree model is created decision trees are added one at a time.
In the beginning a decision tree is fitted to the entire training data. No bagging or other sampling method is performed here.
To enhance the existing tree or tree ensemble it firstly has to be assessed where the weaknesses of the previous decision tree or tree ensemble are. This is done by calculating the *residuals* for every data point. The residual of a data point is the differences of it's prediction to it's real label. The residual r_i for all i in the trainings set can be calculated as follows:

$$r_i = y_i - h(X_i) \tag{2.22}$$

Here y_i is the true label of the data point X_i and $h(X_i)$ is the prediction of the model. Note that this prediction$h(X_i)$ is the probability of the class C_i, as mentioned in the section 'Decision tree'. The residuals that emerge from this process for every data point can be seen as a *weight* for all data points. The higher the residual of a data point, the worse the tree or ensemble of trees was at classifying it correctly. That means the model is weak at data points with high residual. In the next step more attention should be given to data points with high residual, to counteract these weak points.
The next tree will not be fitted to the the initial labels y_i of the training set, but to the residuals of the previous tree. This gives the data points a weight. Data points with higher weight are more relevant to construction of the decision tree. This makes sense because a heigh residual means a weak performance of the previouse decision tree or ensemble. This means for the following constructed decision tree y_i will be set to r_i $(y_i = r_i)$.
Each created tree is added to the tree ensemble and the process is repeated until a stopping criterion is reached, like the maximal number of trees in the ensemble or a minimum threshold of enhancement in performance is undercut in one iteration.
Side note: In a gradient boosted tree algorithm, the weights are not only getting updated by the residuals, but a gradient descent procedure is used to minimize a loss function when adding trees. The details of this will not be discussed for the reason of being beyond the

scope of the theory section of giving a introduction to various methods.

To make predictions on a new data point a majority vote across all decision trees is conducted and the winning class is predicted for the data point. The method reduces variance and minders the risk of overfitting similar to the previous tree ensemble method.

2.3.6 Naive Bayes

In this chapter naive bayes classification will be introduced. The method based on Bayes' theorem. It is also assumed that the effect of an feature value of a data point on a given class is independent of the values of the other feature values of the data point. The goal of a naiv bayes classification is predicting a class by finding the highest of the probability of the probabilities of a given data point belonging to a class. The method primarily used in text classification. A model based on naiv bayes is non-linear.
The method is explained in the general setting for multiclass classification.
In the following we will regard data points A with n features A_1 to A_n. A concrete $X \in A$ with the features x_1 to x_n where n is the number of features of X, hence the length of the feature vector of X, is used. The ith class is denoted as C_i.
The task of finding the probability of an given data point X belonging to a specific class C_i can be expressed as maximising the following expression for every C_I :

$$p(C_i|X) \tag{2.23}$$

The equation 2.22 is the a posteriori probability of C_i conditioned on X.
The initial goal of maximising $p(C_i|X)$ can be rewrote using *Baye's theorem*:

$$p(C_i|X) = \frac{p(X|C_i)P(C_i)}{P(X)} \tag{2.24}$$

$p(X)$ is the a priori probability of X. $p(C_i)$ is the a priori probability of C_i.
The a priori probability of X does not change when comparing probablities of C_i conditioned on X. Therefore the factor can be dismissed in this comparison. This way the task is to maximise only the following term:

$$p(X|C_i)p(C_i) \tag{2.25}$$

The a priori probability of C_i $p(C_i)$ is also known. It can be approximated from the ratio of data points labelled as class C_i in the training data:

$$p(C_i) = \frac{\text{number of data points labelled as} C_i}{\text{number of total data points}} \tag{2.26}$$

This means this term doesn't has to be calculated again.
Still the term

$$p(X|C_i) \tag{2.27}$$

remains. Since X can consist of many features and represents a big feature vector it is computational expensive to calculate $p(X|C_i)$. To simplify the computation involved a assumption is made that, in a sense, may be considered 'naive'.
The assumption made is *class conditional independence*. This means the effect of an feature value of a data point on a given class is independent of the values of the other feature values of the data point. Mathematically that assumption allows the following:

$$p(X|C_i) \approx prod_{k=1}^{n} p(x_k|C_i) \tag{2.28}$$

The probabilities $p(x_1|C_i), ..., p(x_n|C_i)$ can be more easily calculated from the training data: For A_k being continuous-valued the values are assumed to have a Gaussian distribution with a mean μ and standard deviation σ defined by

$$p(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.29}$$

so that for $p(X|C_i)$ holds:

$$p(X|C_i) = p(x, \mu, \sigma) \tag{2.30}$$

This way the mean $\mu_C i$ and the standard derivation $\sigma_C i$ of values for features A_k for training samples of class C_i need to be calculated.

A data point is predicted as a class C_a $in[C_1, ..., C_k$ if and only if C_a maximes $(X|C_i)p(C_i)$ for all classes.

3 Analysis of the pattern recognition problem

3.1 Analysis of the time series data

This chapter will provide a closer examination of the time series data measured by the Texas Intrument's SensorTag. This data is prerequisite for the further work of this paper. The sensor data is measured and collected at a sampling rate of 10 measurements per second. To optimize accuracy, the maximal sampeling rate of the Texas Instrument's SensorTag was chosen. The measurements include acceleration in 3 spacial axis. The axis each correspond to a direction on the Sensorboard and are orthogonal to each other as shown in figure 3.1. The acceleration is measured in g, which refers to the acceleration that is imparted by the gravity of earth. For g the following convertion to m/s^2 holds true: $1g = 9.80665m/s^2$.
The acceleration in any particular direction will be referred to as x, y or z acceleration.

The measurements are taken simultaneously and handled together as one measurement collection. This collection is enriched with the time of the measurement in form of a Unix timestamp.
A measurement collection as described above will also be referred to as a data point. The single measurements (for instance x acceleration) of a measurement collection are the features of a data point.
An exemplary data point in a key-value-pair format can be seen here in figure 3.2. The keys in figure 3.2 stand for acceleration in x direction, acceleration in y direction, acceleration in z direction and the Unix timestamp when the measurement took place.
Using the time of measurement, data can be plotted on a time axis plot as seen in figure 3.3.

Data from the Sensorboard has been collected in isolated experiments. During the experiments of the Sensorboard 69 collisions were simulated. Accordingly, 69 of the 6453 data points collected, were labelled as a collision while the rest was labelled as no collision. The labelling was implemented by analysing video footage, recorded during the experiment. The exact time of a collision in the video was matched with the time of the recorded sensor data to label data points correctly.
The distribution of the data points labeled as collision or no collision can be seen in figure

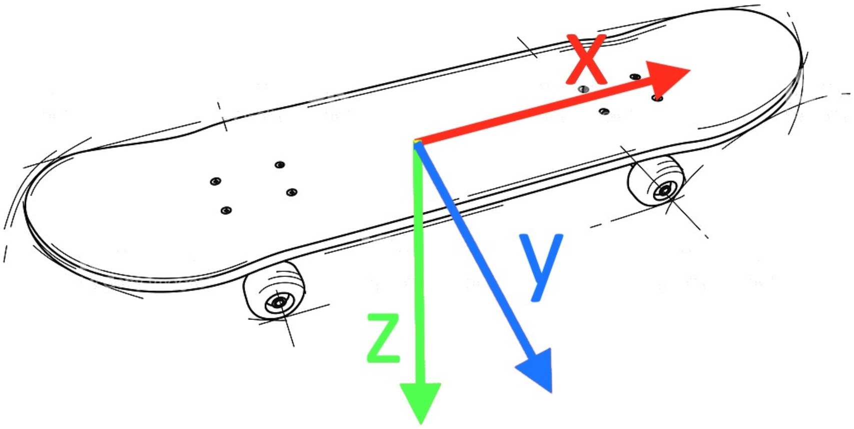


Figure 3.1: Illustration of Sensorboard and axis of measurement. The driving direction is to the right.

3.4. The scatter plot shows the relationship between the x and z acceleration and the label of the data point. The y acceleration is not visualized here. A blue data point represent a collision the red data points represent no collision.

3.2 Solution strategy

Combining the knowledge about the sensor data from the previous section and the objective as described in chapter 1, this chapter will narrow down on a specific problem type and solution strategy. In the process, the limitations described in Chapter 1 have to be considered.

A corpus of sensor data was collected as described above and labelled according to the event of a collision or no collision. In chapter 1 the necessity of using the Apache Spark library module 'MLLib' is constituted.

Having a set of labeled sensor data allows for a supervised machine learning approach. Furthermore the existence of concrete data points which represent the state of the Sensorboard at a given time makes it possible to base the recognition of collisions soly on sinlge data points. The recognition of collisions can then in turn be formulated as the problem of

```
{
  "accx": -0.079590,
  "accy": 0.001465,
  "accz": -1.016602,
  "time": 1502735960911
}
```

Figure 3.2: Example data point in key-value-pair notation notation

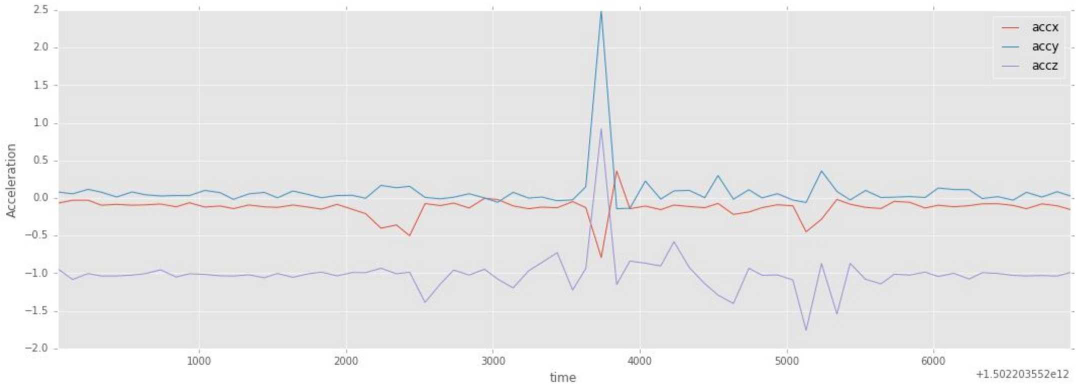


Figure 3.3: Example of measured accelerations on 3 Axis during a collision

deciding weather a given data point corresponds to a collision or no collision. A problem of this kind can be solved by binary classification.
Having specified the solution method to a binary classification approach, specific algorithms can be considered. For the specific task of supervised binary classification the following algorithms are supported by the MLLib module:

- Linear SVM
- Logistic regression
- Decision tree
- Random forest
- Gradient-boosted trees
- Naive Bayes

Only the possible algorithms mentioned here will be parameterized and trained on the collected data in chapter 4.

To assess the performance of resulting models a clear metric will be used for comparison. This metric has to represent the unequal importance of detecting a collision and preventing false alarms. As mentioned in chapter 1 the act of detecting a collision that actually

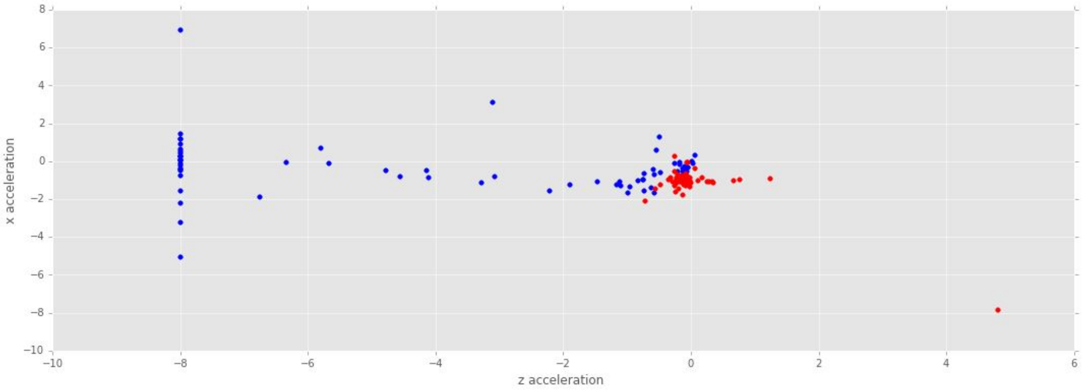


Figure 3.4: Distribution of data points labeled as collision (blue) or no collision (red)

happened (*true positives*) is more important than preventing the detection of a collision that didn't happen (*false positives*). In a classification setting the effectiveness in detecting collision that actually happened is described by recall. The effectiveness of preventing the detection of collision that didn't happen is described by precision. The metric that will be used to represent the unequal relationship between precision and recall is a *f_2-score*. This metric puts more weight on the recall of a model. This way a later evaluation of trained classifiers can be done based on one correctly weighted metric.

4 Algorithm configuration

Many of the methods proposed in chapter 3 require *hyper-parameters.* A hyper-parameter is a setting parameter that varies aspects of the algorithm and might lead to better or worse results.To ensure that all the algorithms are comparable with each other this chapter goes over the process of finding the optimal hyper-parameters for every approach. In the process a f_2-score is assigned to every algorithm under the condition of optimal hyper-parameters.

Hyper-parameter tuning

The process of searching and finding good hyper-parameters for a given algorithm is called *hyper-parameter tuning.* Searching for good hyper parameters takes the following aspects to be considered:

- **an estimator**
  The *estimator* in this case is a binary classifier. The 8 different binary classifiers have to be considered as mentioned in chapter 3.

- **a hyper-parameter space**
  The *hyper-parameter space* defines a set of parameters or combination of hyper-parameters that will be tested. From this set one element will be chosen as most fitting. Because every algorithm requires different hyper-parameters, the hyper-parameter space is different for every algorithm. The hyper-parameter space will be defined in the sections below for every algorithm.

- **a method for searching or sampling candidates**
  The method that is chosen is grid search. Grid search is a brute force search, which uses an exhaustive searching through a defined hyper-parameter space. Models and iteratively trained and evaluated by a score function. The model scores are then compared. The hyper-parameters that produced the best performing model will be chosen as optimal ones for the algorithm.

- **a cross-validation scheme**
  The *cross-validation scheme* used is a k-fold validation, with k=10.

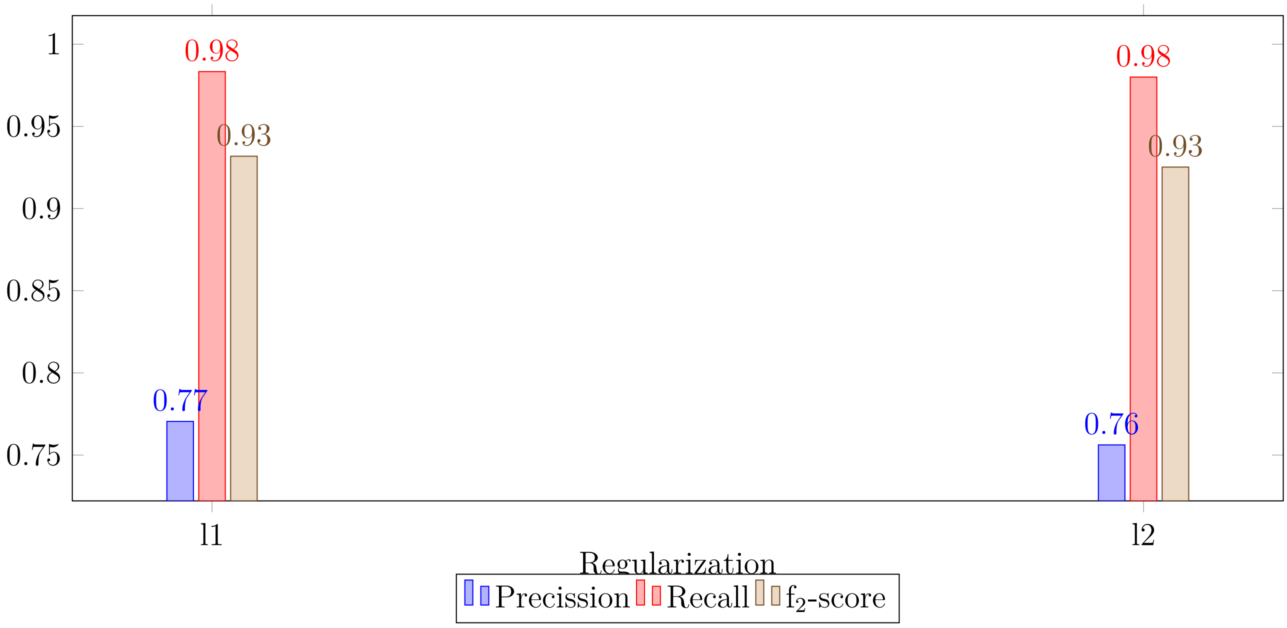
- **a score function**
  As mentioned in chapter 3 a f_2-score will be used to evaluate the models in the end. Our goal for the hyper-parameter tuning is thereby also to maximise the f_2-score. For that reason the f_2-score is used as a score function.

For the hyper-parameter search no external functions or libraries are used. The search algorithm is implemented in the following way.
For the given estimator (one of the possible classification algorithms) the parameter space is defined manually for every hyper-parameter.
Models are now repeatedly trained by iterating through every combination of hyper-parameters in the parameter spaces.
For every trained model a k-fold validation, with k=10, is performed.
During the k-fold validation, the f_2-score is calculated.
After this process is done the hyper-parameters that resulted in the highest f_2-score are chosen.
In the following for each of the relevant algorithms a parameter space will be defined. The results of the hyper parameter search will then be presented.

Linear SVM

The tunable hyper-parameters for the the linear SVM classifier is the regularization.
L1 regularization:
$\sum_{i=1}^{n} |y_\mathrm{i} - f(x_\mathrm{i})|$
y_i is the target value and x_i is the estimated value for y_i.
L1 regularization is a loss function that minimizes the sum of the absolute differences of target and estimated value.
L2 regularization:
$\sum_{i=1}^{n} (y_\mathrm{i} - f(x_\mathrm{i}))^2$
y_i is the target value and x_i is the estimated value for y_i.
L2 regularization is a loss function that minimizes the square of the absolute differences of target and estimated value.

The hyper-parameter search reveals the following results for the two regularization approaches:



The best performing SVM model reaches a maximal f$_2$-score of **0.93** (precision: 0.77, recall: 0.98) with the following hyper-parameter:

Regularization: **l1 regularization**

Logistic regression

The tunable hyper-parameters for the the logistic regression classifier is the regularization.

L1 regularization:

$\sum_{i=1}^{n} |y_\mathrm{i} - f(x_\mathrm{i})|$

y$_\mathrm{i}$ is the target value and x$_\mathrm{i}$ is the estimated value for y$_\mathrm{i}$.

L1 regularization is a loss function that minimizes the sum of the absolute differences of target and estimated value.

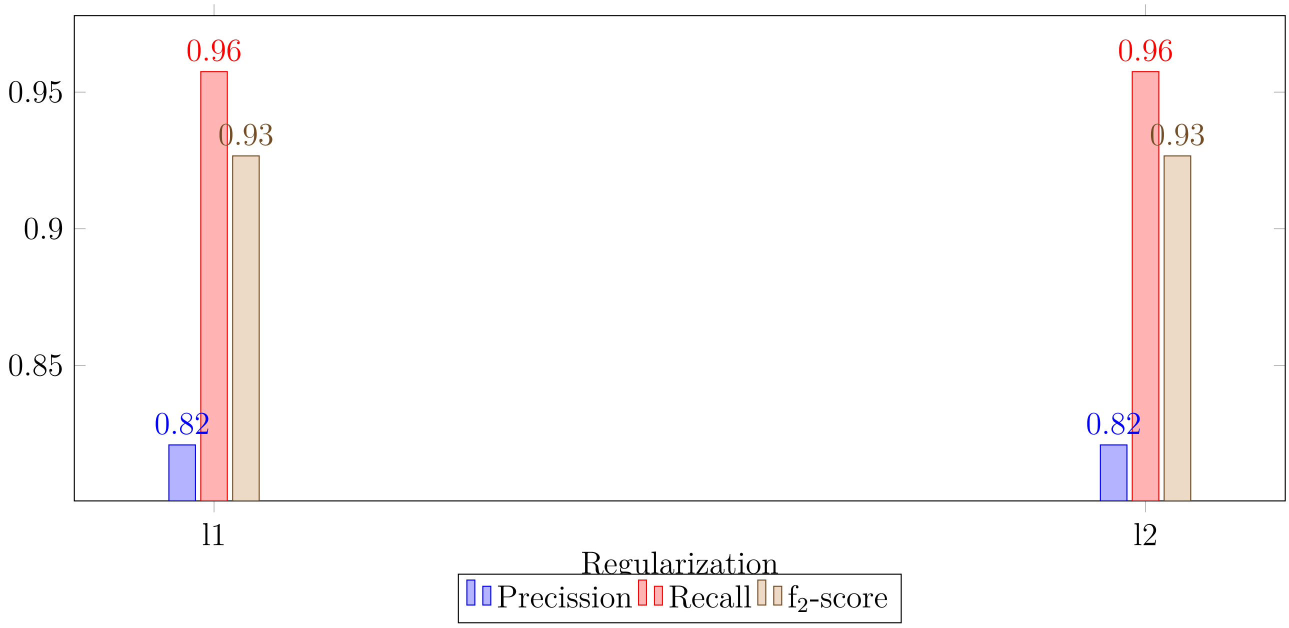
L2 regularization:

$\sum_{i=1}^{n} (y_\mathrm{i} - f(x_\mathrm{i}))^2$

y$_\mathrm{i}$ is the target value and x$_\mathrm{i}$ is the estimated value for y$_\mathrm{i}$.

L2 regularization is a loss function that minimizes the square of the absolute differences of target and estimated value.

The hyper-parameter search reveals the following results for the two regularization approaches:



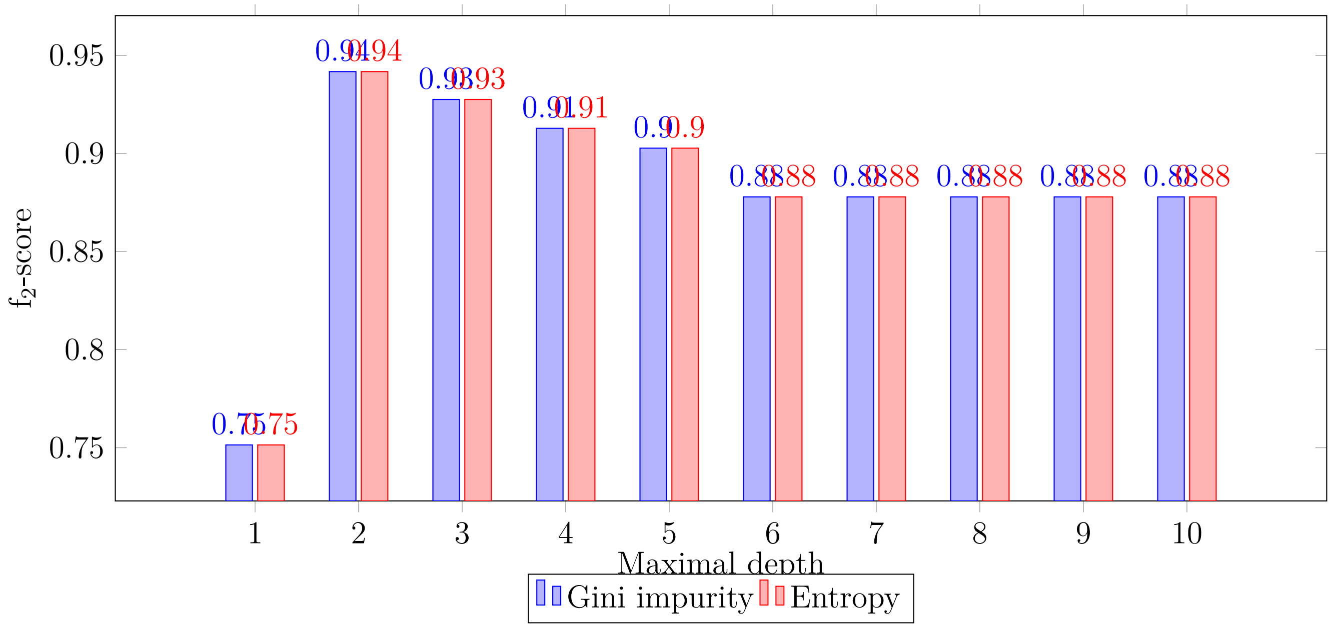
The best performing logistic regression model reaches a maximal f_2-score of **0.93** (precision: 0.82, recall:0.95) independent of which regularization is used.

Decision tree

The tunable hyper-parameters for the decision tree algorithm are:

- **Impurity**
  The impurity measure is used to decide on a splitting rule for each node in the decision tree building process. Two impurity measures for classification are provided (see chapter **??** for more detailed explanation).
  Gini impurity
  Entropy
- **Maximal depth**
  The maximal depth determines a limit of growth for a decision tree. This consequentially limits the number of nodes or decisions that are possible. It represents a stopping criteria.
  The parameter space for the maximal depth is chosen **from 1 to 10** (inclusive).

The hyper-parameter search reveals the following results:



The best performing decision tree model reaches a maximal f_2-score of **0.94** (precision: 0.92, recall: 0.96) with the following hyper-parameters:

Impurity: **Ginni Impurity** or **Entropy**

Maximal depth: **2**

Random forest

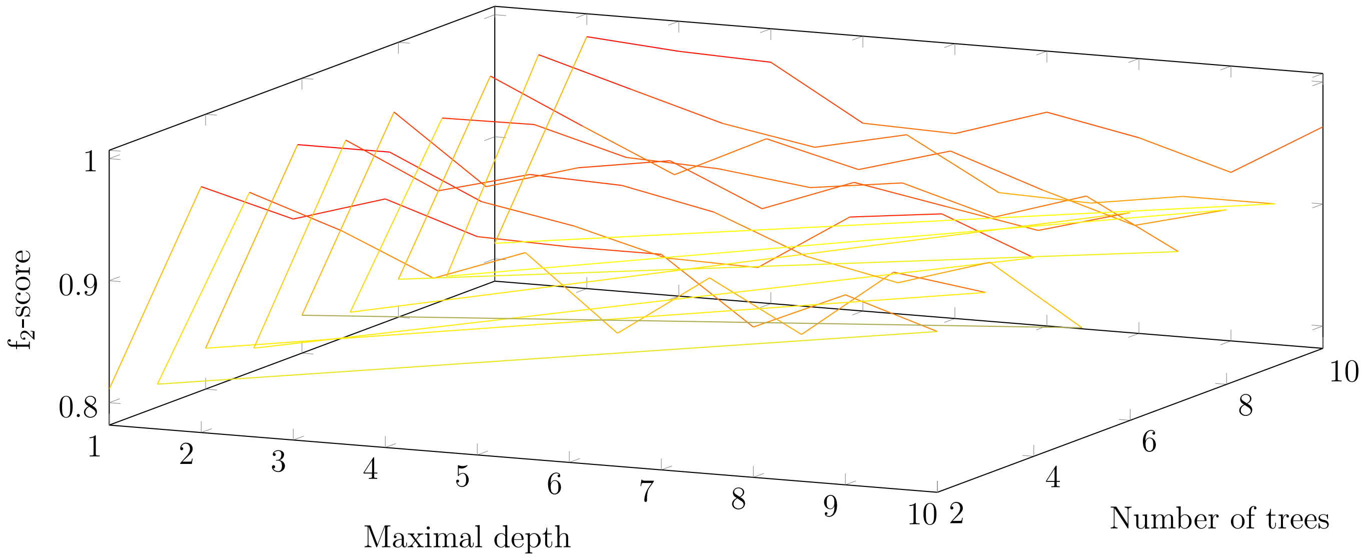
The tunable hyper-parameters for the random forest classifier are:

- **Number of trees**
  The number of trees that are build during the training phase to act as an ensemble of (randomized) decision trees.
  The parameter space for the number of threes is chosen **from 2 to 10** (inclusive).

- **Impurity**
  The impurity measure is used to decide on a splitting rule for each node in the decision tree building process. Two impurity measures for classification are provided (see chapter **??** for more detailed explanation):
  Gini impurity
  Entropy

- **Maximal depth**
  The maximal depth determines a limit of growth for a decision tree. This consequentially limits the number of nodes or decisions that are possible. It represents a stopping criteria.
  The parameter space for the maximal depth is chosen **from 1 to 10** (inclusive).

The hyper-parameter search reveals the following results:



The best performing decision tree model reaches a maximal f$_2$-score of **0.99** (precision: 0.94, recall: 1.0) with the following hyper-parameters:

Number of trees:	**4**
Impurity:	**Entropy**
Maximal depth:	**2** or **3**

or

Number of trees:	**9** or **10**
Impurity:	**Entropy**
Maximal depth:	**2**

Gradient-boosted trees

The tunable hyper-parameters for the decision tree algorithm are:

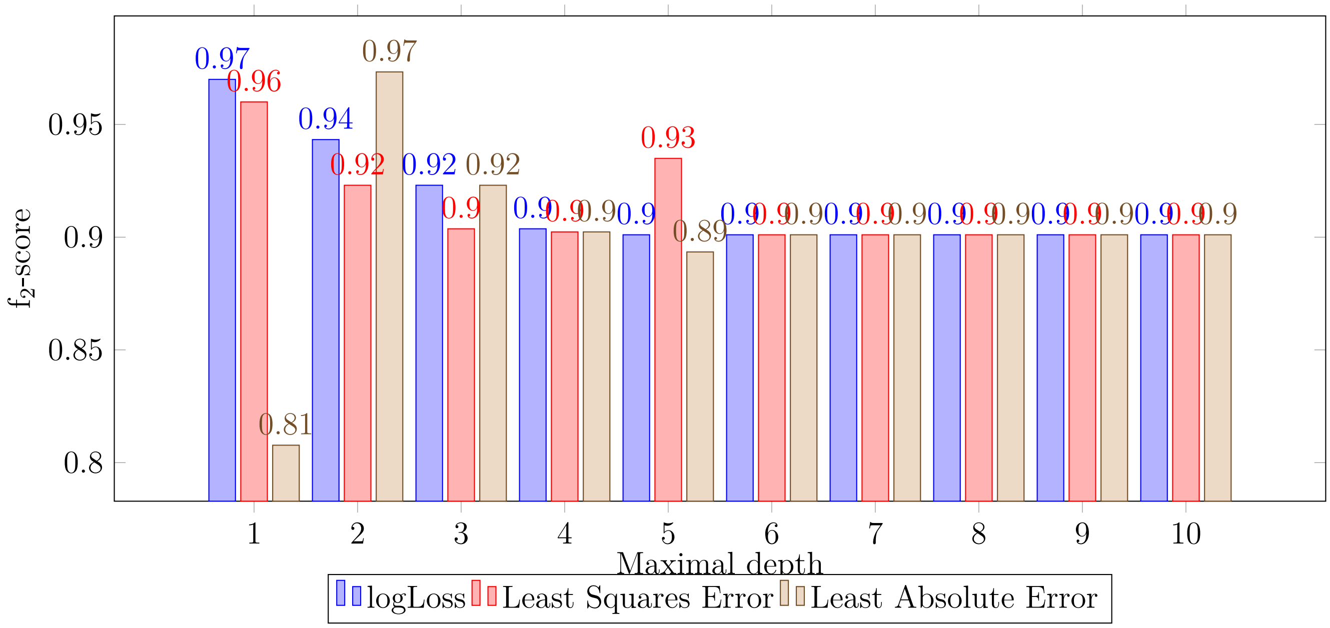
- **Loss function**
  The loss function is used for minimization during gradient boosting. The loss function provided are:
  LogLoss
  Least Squares Error
  Least Absolute Error

- **Maximal depth**
  The maximal depth determines a limit of growth for a decision tree. This consequentially limits the number of nodes or decisions that are possible. It represents a

stopping criteria.

The parameter space for the maximal depth is chosen **from 1 to 10** (inclusive).

The hyper-parameter search reveals the following results:



The best performing Gradient-boosted tree classifier reaches a maximal f$_2$-score of **0.97** (precision: 0.96, recall: 0.98) with the following hyper-parameters:

Loss function: **Least Absolute Error**

Maximal depth: **2**

Naive Bayes

The tunable hyper-parameters for the the Naive Bayes classifier is a λ -value used for additive smoothing. Since the hyper-parameter search with $\lambda = [0.1, 10.0]$ didn't reveal any differences in performance the hyper-parameter is not going to be regarded and will not be discussed further in this paper.

The naive bayes classifier reaches a f$_2$-score of **0.39** (precision: 1.00, recall: 0.34) .

5 Evalutation

The optimal configuration for the different algorithms has been worked out in chapter 4. Now this chapter is devoted to comparing and deciding on an optimal algorithm for the problem of collision detection.
To be able to compare the algorithms they are each used to train a model with the hyper-parameters found in chapter 3. The models are evaluated with a k-fold validation, with k=10. The results can be seen in figure 5.1.

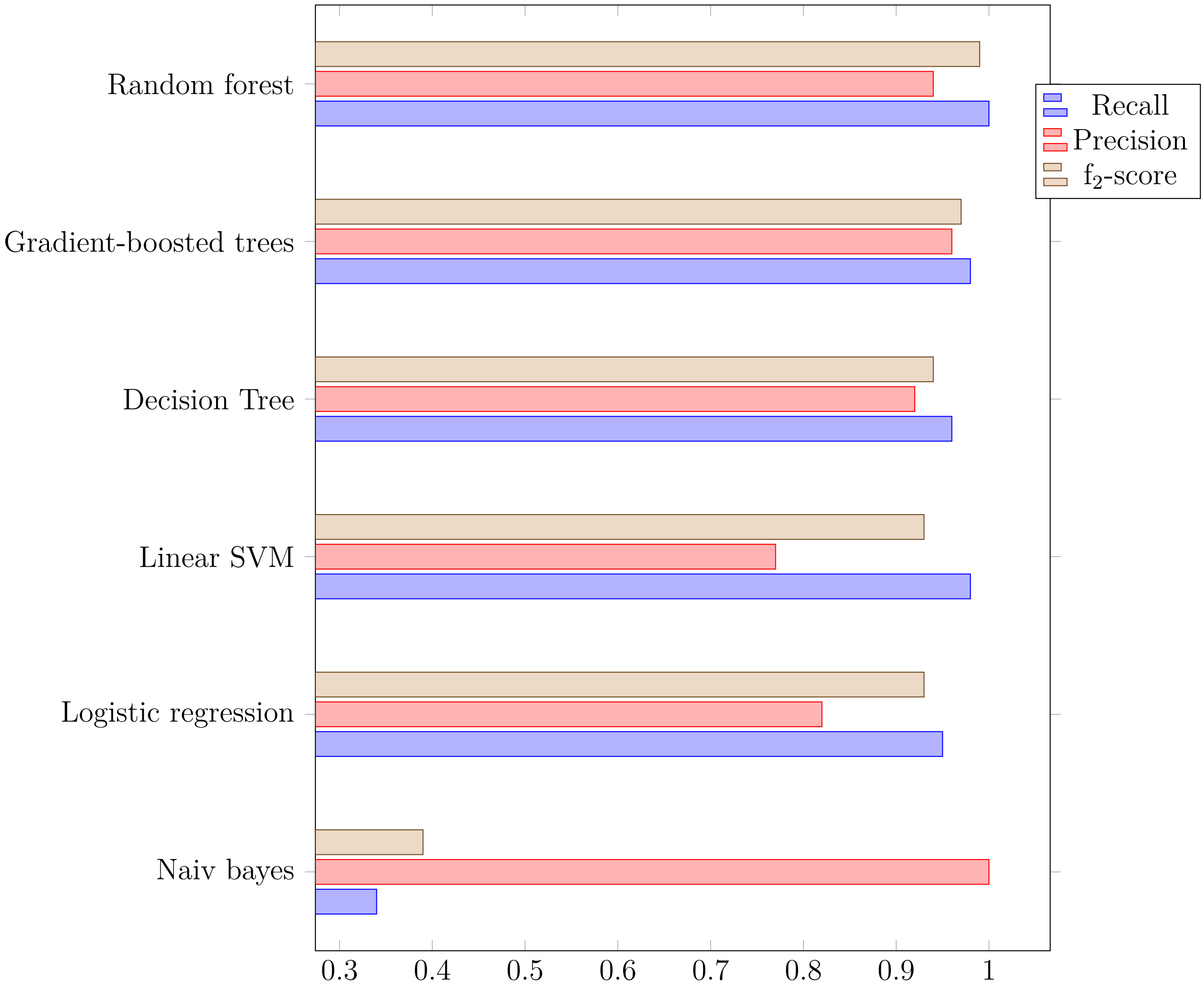


Figure 5.1: Ranking of classification algorithms

According to the f_2-score the best performing method is a random forest classifier with 4 equally well performing hyper-parameter configurations:

Number of trees: **4**	Impurity: **Entropy**	Maximal depth: **2**
Number of trees: **4**	Impurity: **Entropy**	Maximal depth: **3**
Number of trees: **9**	Impurity: **Entropy**	Maximal depth: **2**
Number of trees: **10**	Impurity: **Entropy**	Maximal depth: **2**

The best performing random forest model reaches a maximal f$_2$-score of **0.99** (precision: 0.94, recall: 1.0).

This means a collisions that occur will be classified correctly classified as a collision 100% of the time on the collected data and will be falsely classified as a collision 6% of the time when no actual collision happened.

This results indicate that the machine learning approach to recognizing collisions on a physical board is eligible and can perform well. Still it has to be considered that these results are achieved on data which is generated in experience soly for that reason and results may differ when applied in real world situations.

In figure 5.1 the ranking of the classification methods shows that decision tree based approaches have a advantage over the remaining algorithms. Random forest and gradient-boosted tree models as decision tree assembles performed on average better than a single decision tree. The naiv base algorithm performs worst under the possible algorithms.

6 Reflection and outlook

The previous chapter reached the conclusion that machine learning algorithms are, in fact, feasible for detecting collision detection in the case of the Sensorboard. This shows the potential that machine learning algorithms might have for detecting patterns corresponding to driving behaviours or incidents.

Regardless of the positive result, this study has to be viewed critically. The process of building models and evaluating them is based on datasets generated in experiments in isolated conditions. As no real driving data was recorded in the dataset, the outcome of the research might not be completely transferable to a real driving scenario. Conditions like rough or irregular surfaces and unexpected driving manoeuvres will influence the data quality and therefore the model consistency.

Furthermore the results of this study can not be applied to other driving scenarios as patterns in data will differ based on the vehicle and might not be recognized as easily or with the same methods as described in this paper.

Nevertheless the results of the research clearly indicate potential of this approach for similar use cases. The knowledge gained regarding the feasibility and best methods for a machine learning approach to this use case points further research in the right direction.

In further research the results of this paper can be refined and checked by using sensor data originating from real driving of the Sensorboard as it resembles more realistic driving behaviour. This way the result can be applied to more realistic conditions.

In addition the experimental results of this study can be further examined. It can be studied why some algorithms perform better than the others in the given scenario. The fact that tree based algorithms seem to work better here might be explained by the non linearity of the models created. [itsl p. 319] In contrast the linear SVM and logistic regression represent generate linear models. The superiority of decision tree assembles like the Random forest and gradient-boosted tree over a single decision tree might be explained by a reduction in variance of a classifier and thereby prevention of overfitting and better generalization of the models. A definite explanation for those phenomenon can be explored in further research.

Furthermore the knowledge gained during this research can transposed on other similar problems. Other vehicles like bikes, motorbikes or cars might as well allow for a machine learning based detection of events like collisions.

Even when applied only to a showcase scenario as in this paper, it becomes clear that machine learning applied to IoT data has a lot of potential. Being able to showcase this fact to clients will lead them to a smarter way of making use of sensor data in the information age. In this way the growing field of the Internet of Things in combination with machine learning might have an even bigger impact on industries and consumers. Rolling Window?

Bibliography

[1] Max Meier, *The final theory*, Springer 1999