INSTITUT FÜR INFORMATIK
Computer Vision, Computer Graphics
and Pattern Recognition

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Classification of data from the ATLAS experiments

Michael Janschek

Bachelor Thesis

Beginn der Arbeit:	10. Dezember 2015
Abgabe der Arbeit:	10. MÃd’rz 2016
Gutachter:	Prof. Dr. Stefan Harmeling
	Prof. Dr. Stefan Conrad

Erklärung

Hiermit versichere ich, dass ich diese Bachelor Thesis selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 10. MÃd'rz 2016

Michael Janschek

Abstract

Hier kommt eine ca. einseitige Zusammenfassung der Arbeit rein.

Contents

1 Introduction

This chapter first presents the Higgs Boson Machine Learning Challenge and explains its motivation and goals. It is concluded by an overview of the thesis structure.

1.1 The Higgs Boson Machine Learning Challenge

Kaggle is an internet community of data scientists, it hosts several competitions posed by businesses or organizations. Further services involve open datasets, a "Jobs Board" and "Kaggle Rankings", a scoreboard based on performances of community-members in Kaggles competitions.

1.1.1 Motivation

1.1.2 Goal

> The goal of the Higgs Boson Machine Learning Challenge is to explore the potential of advanced machine learning methods to improve the discovery significance of the experiment. No knowledge of particle physics is required. Using simulated data with features characterizing events detected by ATLAS, your task is to classify events into "tau tau decay of a Higgs boson" versus "background." [higb]

1.2 Overview

In Chapter 2, we will derive the formal problem from the challenges task. We will understand the evaluation metric and finish the chapter by analysing the dataset [higa].

Chapter 3 will use first knowledge about the data to choose simple approaches for classification. After these we will describe several more specific and complex methods.

In Chapter 4 we will use the discussed methods and observe their performance on the challenges data. The thesis closes with a discussion about the approaches and their possible influence on other HEP-applications.

2 Understanding the Challenge

2.1 The formal problem

2.2 The Data

2.3 The Evaluation

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        import math
```

Approximate Median Significance (AMS) defined as: $AMS = \sqrt{2(s+b+b_r)log[1+(s/(b+b_{reg}))]-s}$

where

- $b_{reg} = 10$ is a regulization term (set by the contest),
- $b = \sum_{i=1}^{n} w_i, y_i = 0$ is sum of weighted background (incorrectly classified as signal),
- $s = \sum_{i=1}^{n} w_i, y_i = 1$ is sum of weighted signals (correctly classified as signal),
- log is natural logarithm

```
In [ ]: def calcAMS(s,b):
            br = 10.0
            radicand = 2 *( (s+b+br) * math.log (1.0 + s/(b+br)) -s)
            if radicand < 0:
                print('radicand is negative. Exiting')
                exit()
            else:
                ams = math.sqrt(radicand)
                print("AMS:", ams)
                return ams
```

Following this definition, we can derive a maximum AMS by simply summing the weights of all positive labels.

```
In [ ]: def calcWeightSums(weights,preds,labels):
            s = 0
            b = 0
            for j in list(range(0,len(preds))):
                pred = preds[j]
                label = labels[j]
                weight = weights[j]
                if pred > 0.:
                    if label > 0.:
                        s += weight
                    else:
                        b += weight
            return s,b
```

```
In [ ]: def calcMaxAMS(weights,labels):
            s,b = calcWeightSums(weights,labels,labels)
            ams = calcAMS(s,b)
            print("Found", int(labels.cumsum()[-1]), "signals.")
            print("Weightsums signal:", s, "| background:", b)
            print("Maximum AMS possible with this Data:", ams)
            return ams
```

We generate AMS with good seperable toy-data, starting with the maximum AMS. The data of the actual challenge is weighted to punish wrong-identified signals significantly harder than wrong background. Our toy-data will do so by using its signal-probability as weight, the features are randomized by normal distributions.

```
In [ ]: def generateFeature(label, mu_s, mu_b, sigma_s=5, sigma_b=5):
            if label is 1:
                mu = mu_s
                sigma = sigma_s
            else:
                mu = mu_b
                sigma = sigma_b
            return np.random.normal(mu,sigma)
```

```
In [ ]: def createToyData(n = 100,dim = 3,s_prob = 0.05):
            data= np.zeros(shape = (n,dim),dtype=float)
            if dim < 3:
                print("Operation canceled.",
                      "Data should have at least one",
                      "additional dimension besides weights and labels.",
                      "(dim >=3)")
                return None
            data[:,0] = np.random.rand(n) #weights
            for i in range(0,n):
                if data[i,0] <= s_prob: # label-determination
                    label = 1
                else:
                    label = 0
                data[i,1] = label
                for j in range(2,dim):
                    #mu_s=j*5
                    #mu_b=j*20
                    data[i,j]=generateFeature(label,mu_s=(j-1)*5,mu_b=(j-1)*20)
            return data
```

```
In [ ]: n = 100000
        prob = 0.05
        data = createToyData(n,dim=10,s_prob=prob)
```

```
In [ ]: weights = data[:,0]
        labels = data[:,1]
        calcMaxAMS(weights,labels);
```

We randomly guess labels for a solution for a second AMS with knowledge about the toydatas signal-probability.

```
In [ ]: sol_weights = np.random.rand(n)
        sol = np.zeros(n)
        for i in range(0,n):
            if sol_weights[i] <= prob: # label-determination
                sol[i] = 1
            else:
                sol[i] = 0
```

```
In [ ]: s,b = calcWeightSums(weights,sol,labels)
        calcAMS(s,b);
```

3 Methods of classification

3.1 Logistic regression

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        import math
        from sklearn import linear_model as linMod
```

Data shall have the form of $[w, y, x_1, x_2]$ where

- w is a weight in the intervall $[0, 1)$
- y is the label "0" for "background" or "1" for "signal"
- x_n are randomly generated features with respect to the label

```
In [ ]: def generateFeature(label, mu_s, mu_b, sigma_s=5, sigma_b=5):
            if label is 1:
                mu = mu_s
                sigma = sigma_s
            else:
                mu = mu_b
                sigma = sigma_b
            return np.random.normal(mu,sigma)
```

Approximate Median Significance (AMS) defined as:

$$AMS = \sqrt{2(s + b + b_r)log[1 + (s/(b + b_{reg}))] - s}$$

where

- $b_{reg} = 10$ is a regulization term (set by the contest),
- $b = \sum_{i=1}^{n} w_i, y_i = 0$ is sum of weighted background (incorrectly classified as signal),
- $s = \sum_{i=1}^{n} w_i, y_i = 1$ is sum of weighted signals (correctly classified as signal),
- log is natural logarithm

```
In [ ]: def calcAMS(s,b):
            br = 10.0
            radicand = 2 *( (s+b+br) * math.log (1.0 + s/(b+br)) -s)
            if radicand < 0:
                print('radicand is negative. Exiting')
                exit()
            else:
                return math.sqrt(radicand)
```

```
In [ ]: def calcWeightSums(weights,preds,labels):
            s = 0
            b = 0
```

```
            for j in list(range(0,len(preds))):
                pred = preds[j]
                label = labels[j]
                weight = weights[j]
                if pred > 0.:
                    if label > 0.:
                        s += weight
                    else:
                        b += weight
            return s,b
```

actually generate data

```
In [ ]: #toydata shall have n vectors with 5 dimensions
        n = 100000
        #probability for signal-label
        s_prob = 0.05
        #random values will be used as weights for evaluation later
        weights = np.random.rand(n)
        labels = np.zeros(n)
        x_1 = np.zeros(n)
        x_2 = np.zeros(n)

        for i in range(0,n):
            if weights[i] <= s_prob:
                label = 1
            else:
                label = 0
            labels[i] = label
            x_1[i]=generateFeature(label,mu_s=5,mu_b=20)
            x_2[i]=generateFeature(label,mu_s=5,mu_b=25)
```

visualize

```
In [ ]: %pylab inline
        plt.scatter(x_1, x_2, edgecolor="", c=labels, alpha=0.5)
```

```
In [ ]: def splitList(xList,n):
            aList = xList[:n]
            bList = xList[n:]
            return aList,bList
```

split toydata into training- and testset for the classifier

```
In [ ]: n_train = int(n/10)

        train_x_1,test_x_1 = splitList(x_1,n_train)
        train_x_2,test_x_2 = splitList(x_2,n_train)
        train_labels,test_labels = splitList(labels,n_train)
        test_weights = splitList(weights,n_train)[1]
```

For Comparison, we calculate the best possible AMS
(case: every signal correctly detected)

```
In [ ]: def calcMaxAMS(weights,labels):
            s,b = calcWeightSums(weights,labels,labels)
            ams = calcAMS(s,b)
            print("Maximum AMS possible with this Data:", ams)
            return ams
```

```
In [ ]: calcMaxAMS(test_weights,test_labels);
```

we initialize the Logistic Regression Classifier, shape the input-data and fit the model

```
In [ ]: logReg = linMod.LogisticRegression(C=1e5)

        train_x = np.array([train_x_1,train_x_2]).transpose()
        test_x = np.array([test_x_1,test_x_2]).transpose()
        train_labels = np.array(train_labels).transpose()
        test_labels = np.array(test_labels).transpose()

        logReg.fit(train_x,train_labels)

        logReg.sparsify()

        predProb = logReg.predict_proba(test_x)
        pred = logReg.predict(test_x)
        score = logReg.score(test_x,test_labels)

        print("Score:", score)
```

```
In [ ]: s,b = calcWeightSums(test_weights,pred,test_labels)
        calcAMS(s,b)
```

We successfully tested logistic Regression, now let's use it on actual CERN-Data.

```
In [ ]: import KaggleData;
```

```
In [ ]: csvDict,header = KaggleData.createCsvDictionary()
```

Trainingset has key "t"
Public Testset has key "p" (note: "p" won't work, using private Testset ("v"))

```
In [ ]: def getFeatureSets(featureName):
            trainFeature = KaggleData.getFeatureAsNpArray(
                csvDict,header,featureName,["t"],hasErrorValues = True)
            testFeature = KaggleData.getFeatureAsNpArray(
                csvDict,header,featureName,["v"],hasErrorValues = True)
            return trainFeature, testFeature
```

```
In [ ]: train_eventList,test_eventList = getFeatureSets("EventId")
        train_labels,test_labels = getFeatureSets("Label")
        test_weights = getFeatureSets("KaggleWeight")[1]
```

We observe the relation Label <=> Weight

```
In [ ]: signal_sum = int(test_labels.cumsum()[-1])
        background_sum = int(len(test_labels)-signal_sum)
        signal_weight = 0
        background_weight = 0
        for i in range(0,len(test_labels)):
            if test_labels[i] > 0:
                signal_weight += test_weights[i]
            else:
                background_weight += test_weights[i]
        print(background_weight/background_sum)
        print(signal_weight/signal_sum)
```

We can observe, that False signals will be weighted a lot heavier than True signals.

If a classifier achieved a higher AMS while detecting less signals,
we can make statements about the usabilty of the features, the classifier used.

We choose features with beneficial properties for classifying.

```
In [ ]: (train_DER_met_phi_centrality,
         test_DER_met_phi_centrality) = getFeatureSets("DER_met_phi_centrality")
        (train_DER_pt_ratio_lep_tau,
         test_DER_pt_ratio_lep_tau) = getFeatureSets("DER_pt_ratio_lep_tau")
```

Using DER_mass_MMC was not allowed in the former contest, we use it here anyway to test our classifier

```
In [ ]: (train_DER_mass_MMC,
         test_DER_mass_MMC) = getFeatureSets("DER_mass_MMC")
```

```
In [ ]: train_labels = np.array(train_labels).transpose()
        test_labels = np.array(test_labels).transpose()
```

```
In [ ]: calcMaxAMS(test_weights,test_labels)
        print("True Signals:",int(test_labels.cumsum()[-1]))
```

We start with one feature and add more with every regression to see improvement of the AMS

```
In [ ]: def logisticReg(train_x,train_labels,test_x,test_labels):
            logReg = None
            logReg = linMod.LogisticRegression(C=1e5)
            logReg.fit(train_x,train_labels)
```

```
            logReg.sparsify()
            predProb = logReg.predict_proba(test_x)
            pred = logReg.predict(test_x)
            signals = int(pred.cumsum()[-1])
            print("signals read:", signals)
            if signals is not 0:
                s,b = calcWeightSums(test_weights,pred,test_labels)
                ams = calcAMS(s,b)
            else:
                ams = 0
            print("AMS:",ams)
            return predProb,pred,score
```

```
In [ ]: train_x = np.array(
            [train_DER_met_phi_centrality,
             train_DER_pt_ratio_lep_tau]).transpose()
        test_x = np.array(
            [test_DER_met_phi_centrality,
             test_DER_pt_ratio_lep_tau]).transpose()
        pred = logisticReg(
            train_x,train_labels,
            test_x,test_labels)[1];
        pred.cumsum()
```

```
In [ ]: def logRegFor(fList):
            for feature in fList:
                print("Feature:",feature)
                trainList_x,testList_x = getFeatureSets(feature)
                train_x = np.array([trainList_x]).transpose()
                test_x = np.array([testList_x]).transpose()
                logisticReg(train_x,train_labels,test_x,test_labels)[1];
```

```
In [ ]: (train_PRI_tau_pt,
         test_PRI_tau_pt) = getFeatureSets("PRI_tau_pt")
        (train_DER_met_phi_centrality,
         test_DER_met_phi_centrality) = getFeatureSets("DER_met_phi_centrality")
        (train_DER_pt_h,
         test_DER_pt_h) = getFeatureSets("DER_pt_h")
        (train_DER_pt_ratio_lep_tau,
         test_DER_pt_ratio_lep_tau) = getFeatureSets("DER_pt_ratio_lep_tau")
        (train_DER_mass_transverse_met_lep,
         test_DER_mass_transverse_met_lep) = getFeatureSets("DER_mass_transverse_met_lep")
```

we are able to achieve a higher AMS by adjusting the decision-threshold (around 0.25)

```
In [ ]: def bestThreshold(predProb):
            thresh = 0
            maxAMS = 0
            maxThresh = 0
            for thresh in np.linspace(0.2,1.0,100):
                newPred = np.zeros(len(predProb))
```

```
            for i in range(0,len(predProb)):
                if predProb[i][1] > thresh:
                    newPred[i]=1
            s,b = calcWeightSums(test_weights,newPred,test_labels)
            ams = calcAMS(s,b)
            if ams > maxAMS:
                maxThresh = thresh
                maxAMS = ams
                signals = int(newPred.cumsum()[-1])
        print("Maximum AMS:",maxAMS, "with threshold", maxThresh)
        print("Signals read:", signals)
```

```
In [ ]: train_x = np.array(
            [train_PRI_tau_pt,
             train_DER_met_phi_centrality,
             train_DER_pt_h,
             train_DER_pt_ratio_lep_tau]).transpose()
        test_x = np.array(
            [test_PRI_tau_pt,
             test_DER_met_phi_centrality,
             test_DER_pt_h,
             test_DER_pt_ratio_lep_tau]).transpose()
        (predProb,
         pred) = logisticReg(
            train_x,
            train_labels,
            test_x,
            test_labels)[0:2];
        bestThreshold(predProb)
```

```
In [ ]: train_x = np.array(
            [train_PRI_tau_pt,
             train_DER_met_phi_centrality]).transpose()
        test_x = np.array(
            [test_PRI_tau_pt,
             test_DER_met_phi_centrality]).transpose()
        predProb,pred = logisticReg(
            train_x,
            train_labels,
            test_x,
            test_labels)[0:2];
        bestThreshold(predProb)
```

```
In [ ]: train_x = np.array(
            [train_DER_met_phi_centrality,
             train_DER_pt_ratio_lep_tau]).transpose()
        test_x = np.array(
            [test_DER_met_phi_centrality,
             test_DER_pt_ratio_lep_tau]).transpose()
        predProb,pred = logisticReg(
            train_x,train_labels,
            test_x,
```

```
            test_labels)[0:2];
        bestThreshold(predProb)
```

```
In [ ]: train_x = np.array(
            [train_DER_met_phi_centrality,
             train_PRI_tau_pt]).transpose()
        test_x = np.array(
            [test_DER_met_phi_centrality,
             test_PRI_tau_pt]).transpose()
        predProb,pred = logisticReg(
            train_x,
            train_labels,
            test_x,test_labels)[0:2];
```

3.2 k-nn classification

3.3 The winning methods

3.3.1 Neural networks

3.3.2 Regularized greedy forest

3.3.3 XGBoost

4 Results on the Kaggle data

4.1 k-nn classification

4.2 Comparision of all methods

5 Discussion

References

[ABCG+15] ADAM-BOURDARIOS, Claire ; COWAN, Glen ; GERMAIN, C´ecile ; GUYON, Isabelle ; K´EGL, Bal´azs ; ROUSSEAU, David: *Learning to discover: the Higgs boson machine learning challenge.* http://www.http://opendata.cern.ch/record/329, January 2015. – Version 2.3

[higa] *Dataset from the ATLAS Higgs Boson Machine Learning Challenge 2014*. http://www.http://opendata.cern.ch/record/328, . – Accessed: 2015-12-10

[higb] *Higgs Boson Machine Learning Challenge*. https://www.kaggle.com/c/higgs-boson, . – Accessed: 2016-01-03

List of Figures

List of Tables