Hyperpartisan News Detection

Bachelor Thesis

presented by
Larissa Strauch
Matriculation Number 1518629

submitted to the
Data and Web Science Group
Prof. Dr. Ponzetto
University of Mannheim

Juli 2019

Contents

List of Algorithms

List of Figures

List of Tables

Glossary

CBoW Continous Bag of Words. 3, 4

IDF Inverse Term Frequency. 13

TF Term Frequency. 13

TF-IDF Term Frequency-Inverse Term Frequency. v, 2, 13

Chapter 1

Introduction

1.1 Problem Statement

1.2 Contribution

1.3 Related Work

Chapter 2

Fundamentals

2.1 Term Frequency-Inverse Document Frequency

Term frequency (TF) is a measure that denotes how frequently a term *t* appears a the document *d*. One way to compute TF is

$$tf(t_i, d_j) = \frac{1 + log_{10}(f_{ti,dj})}{1 + log_{10}(\max_{f_{t',dj}:t' \in d_j})}$$

where $1 + log_{10}(f_{ti,dj})$ reflects how many times the term t_i appears in document d_j and $1 + log_{10}(\max_{f_{t',dj}:t' \in d_j})$ the highest occurence of any term in document d_i.

Inverse Doucment Frequency (IDF) points to the assumption that the informativeness of the term *t* is inversely proportional to the number of documents in the collection in which the term appears.

$$idf(t_i) = log10(\frac{|D|}{d' \in D : t_i \in d'})$$

Where $|D|$ is the total amount of doucments in a document set and $d' \in D : t_i \in d'$ is the amout how many times the term t_i appears in the document set.

To compute the weight for the term t_i within the document d_j we simply multiply the *TF* and *IDF* components:

$$w_{ij} = tf(t_i, d_j) \cdot idf(t_i)$$

Therefore, TF-IDF indicates how important a word is to a document in a collection or corpus. It is often used as a weighting factor in Information Retrieval and Text Mining. The TF-IDF value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to control the fact that some words are usually more common than others. (umschreiben?)
TF-IDF is easy to compute. In addition, it is possible to extract the most descriptive

terms, as well as to calculate the similarity between 2 terms. However, TF-IDF is based on the bag-of-words (BoW) model, which is why it disregards aspects such as text position, semantics and co-occurrence.

2.2 Word Embeddings

Word Embeddings are based on the approach of Harris Distributional Hypothesis from 1951, which states, that words that occur in the same contexts tend to have similar meanings.
A Word Embedding provides a word vector for every word. It takes a word and gives it a vector representation by extracting features from that word within the context that word appears in and assigns it a place within a vector space. Two similar word will occupy places which are close to each other within that vector space whereas words that are different, will have locations much further away from each other. This allows computation of distance calculation, which is why, for example, we're able to tell which word is similar to "small" in the same sense as "biggest" is similar to "big". We simply have to compute the vector *X*, which is equal to $Vector_{biggest} - Vector_{big} + Vector_{small}$ and then we find the word which is closest to *X* in the vector space, measured by cosine distance. [Efficient Estimation of Word Representations in Vector Space]

2.2.1 Word2Vec

Word2Vec is a "2-Model Architecture for computing continuous vector representations of words from very large dataset" [Efficient Estimation of Word Representations in Vector Space] that creates an n-dimensional vector space in which each word is represented as a vector. Word2Vecs 2 learning models are the CBoW and Skip-Gram-Model (Figure 3.1).

CBoW uses the context word to predict the target word. The input is a one-hot encoded vector. The weights between the input layer and the output layer can be represented by a $V \cdot N$ matrix *W* where each row of *W* is the *N*-dimensional vector representation v_W of the input word [word2vec Parameter Learning Explained]. The hidden-layer *h* is computed by multiplying the one-hot encoding vector of the input word w_I with the weight matrix *W*.

$$h = W^T_{(k,\cdot)} := v^T_{w_I}$$

Next we have another weight matrix $W' = w'_{ij}$ which is an $N \cdot V$ matrix. With these weights we can finally compute a score u_j for each word in the vocabulary

$$u_j = {v'}^T_{w_j} h$$

where ${v'}_{w_j}$ is the *j*-th column of the matrix *W'*. Afterwards we use *softamax*, which is a log-linear classification model, to obtain the posterior distribution of words.

$$p(w_j|w_I) = y_j = \frac{exp(u_j)}{\sum_{j'=1}^{V} exp(u_{j'})}$$

In contrast to the CBoW model, Skip-Gram uses the target word to predict the context words. The input is still a one-hot encoding vector, the hidden layers definition stays the same as in the CBoW model, each output is still using the same hidden layer to output matrix as in the CBoW model $p(w_{c,j} = w_{O,c}|w_I) = y_{c,j} = p(w_j|w_I) = y_j$ and the function for $u_j = u_{c,j}$ stays the same. However in the output layer, we are now outputting *C* multinomial distributions.

2.3 SVM

2.4 Multinomial Naive Bayes Classifier

2.5 Random Forest Classifier

2.6 Logistic Regression Classifier

Chapter 3

Data

3.1 Data Description

The given data, on which we want to build our model on, was provided by zenodo.org as part of SemEvals Task 4 [Link zu Task hinzufügen]and consists of 2 independent datasets, which in turn have been divided into a GroundTruth-, Training- and Validation set.
The first dataset, recognizable by the term 'byPublisher', reflects the publisher's general bias set forth by BuzzFeed journalists or MediaBiasFastCheck.com beforehand. It consists of a total of 750,000 items, of which 600,000 belong to the Training- and 150,000 to the Validation set.
In return, the second dataset, recognizable by the term 'byArticle', was scrapped by crowdsourcing at hand and therefore consists of only 645 items without a Validation set.

The GroundTruth dataset was provided as an XML File and consists of the features 'article url', 'labeled-by', 'id' and 'hyperpartisan'. In addition, the GroundTruth dataset scrapped 'byPublisher' contains the feature 'bias'.

- Article-url: Contains the article's url.
- Labeled-by: Reflects whether the respective article was labeled 'byPublisher' or 'byArticle'.
- Id: Allocates each article a unique id.
- Hyperpartisan: Displays whether the particular article has been labeled as hyperpartisan or not.
- Bias: Divides the publisher's bias into 'left', 'left-center', 'least', 'right-center' and 'right'.

The Training dataset was as well provided as an XML file and contains the contents of the website of the respective article. In addition, it consists of the features 'article title' 'published-at' and 'id'.

- Article title: Represents the articles title.
- Published-at: Specifies the published date.
- Id: A unique id, which is the same as the corresponding entry in the GroundTruth dataset.

The given Data has been cleaned in advance, therefore no additional steps were necessary.
The main focus of the datasets is on the Hyperpartisan feature, on which we want to classify the articles as this thesis progresses.

3.1.1 Dataset labelled by Publisher

As mentioned above, this dataset consists of a total of 750,000 articles and is divided into a training record consisting of 600,000 articles and a validation set consisting of 150,000 articles. Summarizing these two sets of data, a total of 375,000 were labelled as 'Hyperpartisan' and 375,000 were not – which corresponds to a 50:50 distribution. But even individually, this distribution does not change.

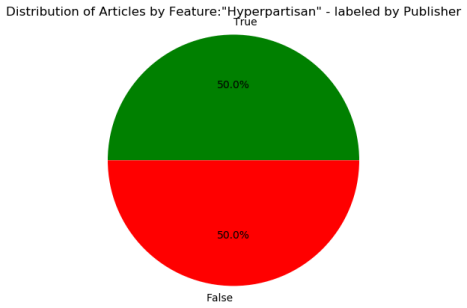
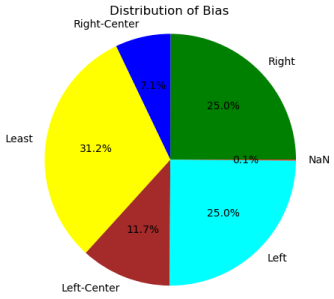


Figure 3.1: Distribution of as Hyperpartisan labelled articles by publisher

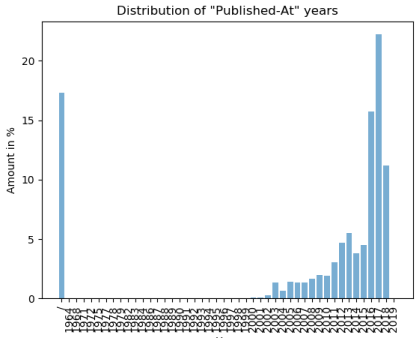
This dataset also includes the feature 'bias', which informs you about the general bias of the publisher. All 375,000 Hyperpartisan labelled all are assigned to either the left or right sectors, but none are right-centre, least or left-centre and are again 50:50 distributed.
The other 50% are split between the remaining bias, with 'Least' owning the largest share at 37%.
The publicity data is distributed over the years 1964-2018, with most of the data coming from 2012-2018.

3.1.2 Dataset labelled by Article

The dataset labelled by Article is a little different to the larger one labelled by publisher. Here the articles were individually labelled by hand. Accordingly, the

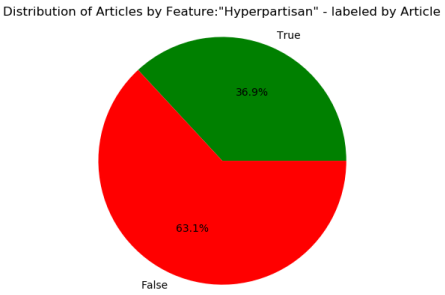


(a) Distribution of Bias

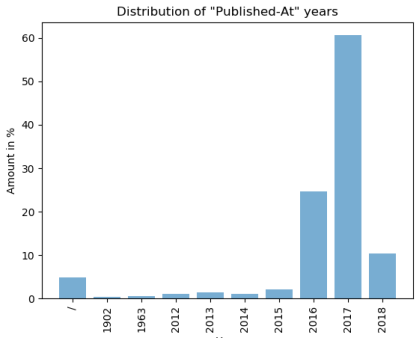


(b) Distribution of publishing years

distributions of this dataset are completely different. This becomes quiet obvious if we look at how the distribution of the Hyperpartisan labelled articles is here. Here



(c) Distribution of Hyperpartisan labelled articles



(d) Distribution of publishing years

we can see that there is no 50:50 distribution left. Only 36.9% were defined here as Hyperpartisan as shown in figure 2.4.
Moreover, in this dataset, the distribution of publication data is not mainly from the years 2012-2018, but in 2016-2018, with the largest number of articles dating back to 2017 at just under 60% as we can see in figure 2.5. Altogether all 645 articles date from the years 1902-2018.

3.2 Data Analysis

Hyperpartisan means ”extremely partisan; extremely biased in favor of a political party.” [definition]. This often materializes in relation to significant political events. As a result, in the following chapter, I will discuss in detail the direct link between the various features, especially the correlations between publication dates and label, as well as the connections between publisher and label.

Publisher	Amount
The Gateway Pundit	17
OpsLens	14
RealClearPolitics	13
New York Post	10
Salon	8

Table 3.1: Publsihers who have published more than 7 Hyperpartisan Articles

Publisher	Bias	Amount
Fox Business	Left	96175
CounterPunch	Left	39832
Mother Jones	Left	36730
Truthdig	Left	25056
Daily Wire	Right	18570

Table 3.2: Publishers who have published more than 10.000 Hyperpartisan articles

The two tables 2.1 and 2.2 list the publisher who produced the highest amount of Hyperpartisan articles in the respective dataset. To keep track, only those publishers who have published more than 7 Hyperpartisan articles in the dataset 'labelled by article' and who have published more than 10.000 hyperpartisan articles in the dataset 'labelled by publisher' are included in the tables. However, a problem here is the dataset 'Labelled by Publisher', since this record, as previously described, has been labelled by the overall bias of the publisher. Meaning, for the further development, we can not include the publishers, since each article of a publishing house has the same label. What I would like to discuss later, however, is the connection between whether or not one of the publishers listed in Table 2.2 published more articles in important political years.

3.3 Data Preparation

In order to be able to work with the existing data in the further course of this project, several preprocessing steps are necessary. In the preprocessation phase of my bachelor's thesis, the data therefore went through the following steps:

1. Read the XML files.
2. Filter out the important information.
3. Merge the Groundtruth- and Training datasets into a csv file.
4. Remove special characters and stop words.
5. Tokenization and stemming of the datasets.

As a result, in the following section, I will go further into detail how I preprocessed my data.

3.3.1 Read the XML files

Since it is difficult to work with the given data in an XML format, it is necessary to bring them into a format with which it'll be easier to work with. The particular

challenge hereby is the size of the dataset labelled by publisher. A standard algorithm for reading XML files is provided by pythons DOM library ElementTree, called “ElementTree.parse”. This method returns an ElemenTree type, which ”is a flexible container object, designed to store hierarchical data structures in memory” [http://effbot.org/zone/element.htm]. Meaning, that this library forms the entire model in the memory which can pose a problem with very large files, such as ours. As a substitute,I therefore use the method “ElemenTree.iterparse”, which can process XML documents in a streaming fashion, retaining in memory only the most recently examined parts of the tree.

3.3.2 Filter out the important information

As mentioned in Chapter 2.1 Data Description, the XML files include various features, which will play an important role in the further course of this work. It is now necessary to filter these features out of the XML format so the can be better worked with later. In order to be able to do this, we pass an algorithm the, by the iterparse method read content, which then passes through this algorithm in a double for-loop and checks for each item of an element in the content which ”key” is currently processed. I then save this in an array, so that the features which have already been parsed, can be used later.

```
def parse_features(content, publisher):
  for event, elem in content:
    for key, value, in elem.items():
      if publisher:
        if key == 'id':
          id_array_publisher.append(str(value))
        elif key == 'published-at':
          published_at_array_publisher.append(value)
        elif key == 'title':
          title_array_publisher.append(value)
      else:
        if key == 'id':
          id_array_article.append(str(value))
        elif key == 'published-at':
          published_at_array_article.append(value)
        elif key == 'title':
          title_array_article.append(value)
      elem.clear()
```

3.3.3 Merge the Groundtruth- and Training datasets into a csv file

Since both, the Groundtruth- and Trainingdata contain important information, it is necessary to merge them into one file. I decided to use Python’s library pandas to write the two files into one csv file. This, and especially Pandas, allows us to read the file more quickly as well as to access individual rows and columns of the merged file in a targeted manner.

```
feature_extraction.parse_features(xml_training, publisher)
groundtruth_parser.parse_groundtruth(xml_gt, publisher)
content = content_parser.parse_content(content_training)

a_id = feature_extraction.get_id_array(publisher)
published = feature_extraction.get_published_at_array(publisher)
title = feature_extraction.get_title_array(publisher)
bias = groundtruth_parser.get_bias_array(publisher)
hyperpartisan = groundtruth_parser.get_hyperpartisan_array(publisher)

columns = {"ArticleID": a_id,
           "PublishedAt": published,
           "Title": title,
           "Bias": bias,
           "Content": content,
           "Hyperpartisan": hyperpartisan}

tf = Pd.DataFrame(columns)
tf = tf[['ArticleID','Published', Title','Bias','Content','Hyperpartisan'
    ]]
tf.to_csv(titlecsv,encoding='utf-8',index=False)
```

3.3.4 Remove special characters and stop words

Now that we have merged both files and written them into a csv file, we need to remove special characters and stop words. Especially the removal of stopwords is necessary since not all words presented in a document, such as auxiliary verbs, conjunctions and articles [TextClassificationUsingMachineLearning] are useful for training a classifier. It also decreases our corpus which makes it easier to classify later on. (Warum?)

```
stop = stopwords.words('english')

df['Content'] = df['Content'].apply(lambda x: ''.join([item for item in x.
    split() if item not in stop]))
df['Title'] = df['Title'].apply(lambda x: ''.join([item for item in x.
    split() if item not in stop]))
df['Content'] = df['Content'].map(lambda x: re.sub(r"[^a-zA-Z0-9]+", '', x
    ))
df['Title'] = df['Title'].map(lambda x: re.sub(r"[^a-zA-Z0-9]+", '', x))
```

Here it becomes obvious that using pandas was a good choice, as we can now specifically access the 'Content' and 'Title' columns in order to perform this step of preprocessing on only these two and not the whole file.

3.3.5 Tokenization and Stemming of the datasets

After cleaning the dataset, the words are tokenized in order to convert them into numerical vectors so that a classifier is able to work with them. Tokenization is definied as "The process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input".
Stemming is the procedure of reducing the word to its grammatical (morpho-

syntatctic) root. The result is not necessarily a valid word of the language. For example, ”recognized” would be converted to ”recogniz”. Still, the basic word almost always contains the very meaning of the word. Stemming is advantageous in that the algorithm used later now only has to fall back on a few different words and not many, all of which have the same meaning.

In order to implement Stemming and Tokenization, only 2 lines of python code are necessary, due to the Pandas dataframe.

```
df["Content"] = df["Content"].apply(nltk.word_tokenize)
df['Content'] = df['Content'].apply(lambda x: [stemmer.stem(y) for y in x
    ])
```

This will transform our output file in the following way:

Chapter 4

Methodology

The main process for Text Classification includes 7 steps of which the first 4 have already been performed in Chapter 2.

1. Read the Document.
2. Tokenize Text.
3. Stemming.
4. Delete Stopwords.
5. Vector Representation of the Text.
6. Feature Selection and/or Feature Transformation.
7. Learning Algorithm.

As already mentioned in Chapter "Introduction", we build our classifier model by using BERT-Embeddings. To compare how our model performs, in this chapter, I will discuss the classic methods and algorithms that I have used in order to achieve this comparison.

4.1 Vector Representation of the Text

In order for our classifier to be able to work with the text, we first need to transform our words into a feature vector representation. A document is a sequence of words [Text Categorization with Support Vector Machines] so a document can be presented by a One-Hot encoded vector, assigning the value 1 if the document contains the feature-word or 0 if the word does not appear in the document. [TextClassificationUsingmachineLearning]. However, using this technique for word representation, resolves in a $V \cdot V$ Matrix, as we have V-diemensional vertor for each out of V words which can lead to huge memory issues. In addition this does not

notion similiarity between words. Therefore I will go into further detail for better approaches in the next 2 subchapters.

4.1.1 Term Frequency-Inverse Document Frequency

A comparative approach I used in the course of my Bachelor Thesis is Term Frequency-Inverse Term Frequency. By using TF-IDF, we're able to represent our word as a vector by assigning it weight which is computed thorugh Term-Frequency devided by Inverse-Term-Frequency. Python's library *sklearn* provides two ways to implement TF-IDF without having to program TF and IDF by itfself. In order to get a generally better overview, I will now explain the 2-step implementation, but also briefly explain how both steps can be combined in one.

First of all, we have to import CountVectorizer and TfidfTransformer from *sklearn*, as they will later be responsible for representing our words as vectors. As exmplary input data we take the first article of the byArticle csv file.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
```

```
df = pd.read_csv('C:\\Users\\Larissa\\Documents\\Uni\\
    Bachelorarbeit\\Git\\Data\\Preprocessed_ByArticle.csv',
    encoding='utf-8')
content= [df.Content[1]]
```

```
['[donald, trump, ran, many, braggadocios, largely, unrealistic, campaign,
    promises, one, promises, best, hugest, competent, infrastructure,
    president, united, states, ever, seen, trump, going, fix, every,
    infrastructure, problem, country, make, america, great, process,
    unless, brown, american, case, even, massive, natural, disaster, like,
    hurricane, maria, puerto, rico, debt, puerto, rican, citizens,
    government, would, responsibility, nothing, using, federal, emergency,
    disaster, funds, save, lives, american, citizens, infrastructure,
    certainly, mess, point, category, 5, hurricane, ripped, island, 84,
    percent, puerto, rican, people, currently, without, electricity,
    emergency, efforts, hurricanes, irma, harvey, reportedly, went, well,
    trump, praised, well, even, saw, disastrous, approval, ratings, tick,
    slightly, result, however, insufficient, response, ...]']
```

For computing TF as the first step we use *CountVectorizer*'s method *fit_transform()* which now converts the text document into a matrix of token counts.
The method simply counts how many times a term t_i appears in a document d_j.

$$tf(t_i, d_j) = |(tf_i, d_j)|$$

```
count_vect = CountVectorizer()
content_counts = count_vect.fit_transform(content)
content_counts
```

```
<1x84 sparse matrix of type '<class 'numpy.int64'>'
with 84 stored elements in Compressed Sparse Row format>
```

Any term found during this method is thereby assigned a unique index corresponding to a column of the regressive matrix which can be obtained by the method

toarray().

```
content_counts.toarray()
```

```
array([[1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2,
        2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 3, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3, 1, 1, 1, 1, 1, 1,
        2, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 2, 1, 1, 1]],
        dtype=int64)
```

For the second step of comptuting IDF and TF-IDF we use the *TfidfTransformer*'s method *fit_transform()* which converts *content_counts* into TF-IDF weights.

```
tfidf_transformer = TfidfTransformer()
content_tfidf = tfidf_transformer.fit_transform(content_counts)
```

```
array([[
0.0860663 , 0.0860663 , 0.17213259, 0.0860663 , 0.0860663 ,
0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 ,
0.0860663 , 0.17213259, 0.0860663 , 0.0860663 , 0.0860663 ,
0.0860663 , 0.17213259, 0.0860663 , 0.0860663 , 0.0860663 ,
0.0860663 , 0.17213259, 0.17213259, 0.0860663 , 0.0860663 ,
0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 ,
0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 , 0.17213259,
0.0860663 , 0.25819889, 0.0860663 , 0.0860663 , 0.0860663 ,
0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 ,
0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 ,
0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 ,
0.0860663 , 0.0860663 , 0.0860663 , 0.17213259, 0.25819889,
0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 ,
0.0860663 , 0.17213259, 0.0860663 , 0.0860663 , 0.0860663 ,
0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 ,
0.25819889, 0.0860663 , 0.0860663 , 0.0860663 , 0.0860663 ,
0.17213259, 0.0860663 , 0.0860663 , 0.0860663 ]])
```

Term	Average TF-IDF Weight
trump	0.091165
bannon	0.065129
president	0.061829
kimmel	0.052415
money	0.051647
americans	0.049132
people	0.047342
obamacare	0.042512
wall	0.042117
control	0.039068

Table 4.1: Top 10 terms by average TF-IDF weight (By Article)

Terms	Average TF-IDF Weight
balls	0.057251
said	0.051183
medicare	0.047450
martin	0.046519
school	0.044264
university	0.043572
says	0.042079
trump	0.041235
carrier	0.040903
degree	0.040704

Table 4.2: Top 10 terms by average TF-IDF weight (By Publisher)

4.1.2 Word Embeddings

Word2Vec

4.2 Feature Selection and/or Feature Transformation

4.3 Learning Algorithms

4.3.1 SVM

4.3.2 Multinomial Naive Bayes Classifier

4.3.3 Random Forest Classifier

4.3.4 Logistic Regression Classifier

Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Master-/Bachelorarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich bin mir bewusst, dass eine falsche Er- klärung rechtliche Folgen haben wird.

Mannheim, den 31.08.2014 Unterschrift