FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Seminararbeit

Automatische Qualitätssicherung von Anforderungen

Eindeutigkeit – Möglichkeiten und Grenzen automatischer Erkennung

Autor:	Michael Wang
Betreuer:	Dr. Henning Femmer
Datum:	09. Juli 2018

Abstract

The quality of a software system is tightly coupled to the quality of its requirements. One important criterion for high quality requirements is unambiguity.

The purpose of this report is to evaluate to which extent (attachment) ambiguity in software requirements specifications can be automatically detected by comparing the output of different dependency parsers.

We first implement a natural language processing pipeline so that a single requirement can be processed by different dependency parsers. We start by using the *NLP4J* framework for linguistic pre-processing tasks, such as sentence splitting, tokenization, and part-of-speech tagging. We then select and implement three dependency parsers: *NLP4J* parser, *MaltParser*, and *MSTParser*. The result is that three outputs are produced from one requirement.

Afterwards we evaluate the implemented prototype in a case study using the *KeePass* study object. We first evaluate to which extent inconsistiencies between the three parsers occur. For this we focus on prepositions which are the cause of attachment ambiguity and we measure the *Unlabeled Attachment Score*. The result, i.e. the percentage of consistently parsed prepositions, ranges from 69.64% to 76.89%.

Finally we evaluate the remaining roughly 25% to 30% of recognized prepositions that are not consistent between parsers. Out of the first 25 such occurrences, 21 were due to erroneous parsing, but four were due to ambiguity.

In conclusion we find that (attachment) ambiguity can be detected by comparing the output of dependency parsers. We suggest that similarly to how spell-checking tools work, this approach can help in implementing "ambiguity-checking" tools. While a human would still have to evaluate whether ambiguities exist or not, such a tool would still be of aid for requirements writers.

Contents

1 Introduction

Software requirements specifications (SRS) are the starting point for every software engineering project. A high quality of these requirements is both desired and necessary to ensure high software quality. One important criterion for high quality is unambiguity, so that requirements are clear for human readers [16].

These aspects are exemplified in both Glass' law (1997) and Boehm's first law (1975): "requirement deficiencies are the prime source of project failures" [15] and "errors are most frequent during the requirements and design activities, and are the more expensive the later they are removed" [3].

1.1 Motivation

In this report we research to which extent a requirement can be automatically assessed for ambiguities. From this wide area of research, we emphasize the following aspects:

First, we know that one type of ambiguity is structural ambiguity and one form of this can be attachment ambiguity, which is further explained in Section 2.1. Secondly, we know that the structure of a phrase can be automatically assessed by dependency parsers. Lastly, we know that if different interpretations of a phrase are valid due that phrase being ambiguous, then different outputs may occur from different dependency parsers and each such output may be valid.

These three aspects are the motivation for this report. Our goal is to combine these aspects so that we can evaluate whether attachment ambiguity can be detected by comparing the output from different dependency parsers.

1.2 Approach

With this goal in mind, we first implement a natural language processing pipeline that can process a SRS with different dependency parsers. For this purpose we need to first handle the linguistic pre-processing of the SRS, such as sentence splitting, tokenization, and part-of-speech tagging. We then select and implement three different dependency parsers. The result is that three outputs are produced from one SRS.

Afterwards we evaluate this prototype in a case study based on a study object from the industry, namely *KeePass*. In this study we first answer to which extent inconsistencies between the dependency parsers occur. We then evaluate whether these inconsistencies are due to erroneous parsers or due to ambiguities in the SRS.

1.3 Report Outline

In this report we evaluate to which extent ambiguities in SRS can be detected by comparing the output of different dependency parsers. In Chapters 1 and 2 we explain why this is of interest and explain some background knowledge. We then implement a prototype in Chapter 3. This prototype is evaluated in an case study in Chapter 4. Finally, in Chapter 5, we summarize the learnings of this report and present future work.

2 Fundamentals

This chapter introduces background knowledge for later chapters. We begin with an overview on ambiguity and then introduce relevant frameworks and tools.

2.1 Ambiguity

There are several fields of study from which definitions of ambiguity can be derived. One such field is (computational) linguistics. Barry et al. (2003) [2] derive four broad classes of linguistic ambiguity: lexical, syntactic, semantic, and pragmatic ambiguity.

They further categorize each class into multiple variants. Syntactic ambiguity can for example be analytical, attachment, coordination, or elliptical ambiguity.

This report focuses on attachment ambiguity. An example of this is the fictional phrase "*KeePass restricts users with root access*". In this example, the attachment of the preposition "*with*" is ambiguous as it can either modify the noun "*users*" or the verb "*restricts*". This leads to two different interpretations: either KeePass has root access to restrict users, or users with root access are restricted by KeePass.

2.2 Frameworks and Tools

This section introduces two frameworks and tools that are relevant to this report.

We first use *DKPro Core* [13] for our implementation. It is a collection of software components addressing linguistic pre-processing tasks, such as sentence splitting, tokenization, and part-of-speech tagging. It also enables us to select different dependency parsers as part of the processing pipeline [12]. In this report, version 1.9.2 is used to process a single SRS with three different dependency parsers.

We also use *DependAble* [5]. This toolset provides visualization and evaluation tools for dependency parsing [8]. It aids us in calculating metrics for our evaluation.

3 Implementation

In this chapter we describe the implementation of a natural language processing pipeline in which we input a SRS and output the result from three different dependency parsers. We start by explaining the linguistic pre-processing: input, tokenization, and part-of-speech tagging. We then examine the three dependency parsers used.

3.1 Linguistic Pre-Processing

This section explains the linguistic pre-processing of a SRS. We use the *DKPro Core* framework introduced in Section 2.2 for this task.

XML Input

The first component in our processing pipeline is *XMLTextReader* [11] and it is used to read in a SRS in the *.xml* format. The selection of this SRS, i.e. the study object, is described further in Section 4.2.

NLP4J Tokenizer and POS tagger

Next in the pipeline are the *Nlp4JSegmenter* [10] and *Nlp4JPosTagger* [9] components from the *NLP4J* [21] toolkit. They tokenize the output from the previous component and then assign part-of-speech (POS) tags to each token. We select the *NLP4J* framework because of three reasons: First, *NLP4J* is already familiar to us. Secondly the developers of *NLP4J* and the *DependAble* toolkit explained in Section 2.2 are the same. Most important however is that the framework is both accurate and fast to use. Choi (2016) evaluated the POS-tagger and achieved a 97.64% accuracy on the Wall Street Journal corpus [4]. The tagger can also process over 82.000 tokens per second on an Intel Xeon 2.30GHz machine [22].

3.2 Dependency Parsers

This section presents the three dependency parsers used. These parsers were selected mainly based on their output format, that is to which extent their dependency labels are comparable. An example for this is that a *prepositional modifier* is labeled as *prep* in all three parsers. Each parser was provided with the same input fro the pre-processor detailed in Section 3.1.

NLP4J Dependency Parser

The *NLP4J* dependency parser uses a "transition-based, non-projective parsing algorithm" [20]. We select this parser because we want to avoid compatibility issues as previous components from the linguistic pre-processing steps are also from the *NLP4J* toolkit. We also select this parser because it is both accurate and fast to use. It achieves an accuracy of 92.26% on the Wall Street Journal corpus for the greedy parser [20] and can process over 14.000 tokens per second on an Intel Xeon 2.30GHz machine. The *NLP4J* parser can output 44 different dependency labels. This set of labels is known as *NLP4J typed dependencies*.

MaltParser

MaltParser is "a system for data-driven dependency parsing" [17]. We select this parser because it outputs *Stanford typed dependencies* [7] which are very comparable to the *NLP4J* labels. Additionally, *MaltParser* can achieve state-of-the-art accuracy but optimization is needed to do so [17]. We simply want a "decent robust dependency parser" [18] and therefore select the *engmalt.poly-1.7.mco* model. We do not optimize *MaltParser* any further, but this could be done if necessary.

MSTParser

MSTParser is "a non-projective dependency parser that searches for maximum spanning trees over directed graphs" [19]. We select this parser because it also outputs *Stanford typed dependencies* and can therefore be easily compared to both *MaltParser* and *NLP4J* parser. We need to use the *eisner 20100416.2* model of *MSTParser* because it is the only model supported by *DKPro Core* that outputs our desired *Stanford typed dependencies*.

4 Evaluation

This chapter presents our case study and evaluates the prototype implemented in Chapter 3. We first explain the design of this study and formulate research questions. This is followed by an overview of the study object used. We then describe the study procedure. Afterwards, we answer the research questions and present the study results. Finally, we discuss these results.

4.1 Study Design

In this section we describe the design of this case study, which is based on the Goal-Question-Metric approach [1].

Goal

The goal of this study is to evaluate to which extent ambiguities can be detected by comparing the output of different dependency parsers. We specifically aim at attachment ambiguity and evaluate the prepositions of a sentence which are the cause of attachment ambiguity as exemplified in Section 2.1.

To accomplish this goal, we compare the output of different dependency parsers. In this comparison, two possibilities can exist as each preposition can be either parsed consistently or it can not be. This is why we first evaluate to which extent the output of dependency parsers is consistent for prepositions.

Assuming that an inconsistency was detected, then two possibilities exist again as this finding can be either due to parsing errors or due to ambiguity. In the first case, a preposition is unambiguous and therefore should be parsed consistently by all parsers. In the second case, a preposition is ambiguous and therefore may be parsed differently by each parser. We evaluate to which extent this second case is true.

Questions

In order to achieve this goal, we formulate the following research questions (RQ):

RQ1 How consistently are prepositions parsed?

The answer to this question analyzes how many findings, i.e. differences between dependency parsers, need to be evaluated for ambiguity. The optimum would be to have perfectly consistent output, i.e. dependency parsers are 100% accurate and there are no ambiguities present in the SRS. Unfortunately this is unrealistic so we want to evaluate to which extent our prototype approximates such a perfect system and SRS. Findings that are inconsistent will need to be evaluated further.

RQ2 To which extent are parsing differences caused by ambiguity?

This question evaluates to which extent findings are caused by ambiguities. A finding can be caused due to two reasons: Either the finding is unambiguous and should be parsed consistently, but the parser is erroneous. Alternatively the finding is ambiguous and the parsers may produce different outputs that are all valid.

Metrics

The following metrics are used to answer the research questions:

RQ1 How consistently are prepositions parsed?

We measure the *Unlabeled Attachment Score (UAS)* of the prepositions.

Possible values range between 0% and 100%, calculated by the ratio of prepositions that were recognized consistently to the total amount of recognized prepositions. This means that we only measure on the precision of the parsers and not on recall. A higher *UAS* is better, as it implies more accurate dependency parsing and a less ambiguous SRS.

The *UAS* is only measured on whether the arc of a predicted dependency was assigned correctly. The label is not taken into account because we would need to compare across different dependency label sets.

RQ2 To which extent are parsing differences caused by ambiguity?

Parsing differences, i.e. findings, are boolean as they are either caused by erroneous parsers or due to ambiguity. We count the number of occurrences for each possibility and also present this as a percentage.

4.2 Study Object

This section presents the study object evaluated in this case study. We examine the *Software Requirements Specification for KeePass Password Safe*, version 1.10 from February 17th 2008. This document is available as part of the *PURE (PUblic REquirements)* data-set [14]. In this data-set, the *KeePass* specification is categorized as follows: It is in the *.pdf* format, has 29 pages, few pictures, is on a high level, is structured, and is from the industry [23]. Our linguistic pre-processing pipeline detailed in Section 3.1 produces 6.843 tokens in 253 sentences from this SRS.

We selected this SRS because of three reasons: First, the document is understandable because it is about a familiar topic, i.e. password management. Secondly, the length of the specification is of a size that can be reasonably handled. Lastly, the SRS was already manually transcribed into the *.xml* format and is thus easier to read into our natural language processing pipeline.

4.3 Study Procedure

In this section we explain the study procedure for both RQ1 and RQ2.

RQ1: How consistently are prepositions parsed?

We only evaluate prepositions for RQ1, thus the first step is to select the relevant tags and labels. Of interest are the *IN* POS-tags, i.e. prepositions, and the *prep* dependency labels, i.e. prepositional modifiers.

Afterwards we use the evaluation tool of *DependAble* to calculate the *Unlabeled Attachment Score (UAS)* of *IN* and *prep* respectively. The amount of *IN* POS-tags is the same for each parser as they use the same input. The amount of *prep* labels produced by a parser can however vary.

The *DependAble* evaluation tool only allows one-to-one comparisons and thus we need to evaluate three sets of combinations to gather the *UAS* for POS-tags. For dependency labels we need to evaluate six combinations because the order matters.

RQ2: To which extent are parsing differences caused by ambiguity?

In RQ2 we evaluate to which extent parsing differences detected in RQ1 are caused by ambiguity. For this purpose we use the comparison tool of *DependAble* to visualize the dependencies produced by each parser.

We evaluate the result manually as follows: First we search for prepositions, looking for *IN*-tags and *prep*-labels. We then check these tags and labels manually and confirm whether the three parsing outputs are consistent. If they are not, then this finding is categorized as one of two possibilities: Either the parser is inaccurate or there is an ambiguity present.

Due to reasons of scope we only evaluated the first 25 of such occurrences.

4.4 Results

This section presents the results of this study which we will then discuss in Section 4.5.

RQ1: How consistently are prepositions parsed?

We first aggregate the distribution of both *IN*-tags and *prep*-labels in Table 1. Each parser outputs the same 527 *IN*-tags as these were assigned by the same POS-tagger in a previous step of the processing pipeline. For *prep*-labels however, the output differs as each parser processes these on their own.

We then present the *UAS IN*-tag distribution in Table 2 and the *prep*-label *UAS* distribution in Table 3. This metric ranges from 69.64% to 76.89% and represents the percentage of the recognized prepositions that are done so consistently. An example for such is presented in Figure 1 which is a very simple prepositional construction. For reasons of clarity we only show the relevant dependency arcs.

	IN	*prep*
NLP4J	527	463
MaltParser	527	450
MSTParser	527	465

Table 1: Distribution of *IN*-tags and *prep*-labels

NLP4J & MaltParser	71.16%
MaltParser & MSTParser	69.82%
MSTParser & NLP4J	69.64%

Table 2: Distribution of *IN*-tag *UAS*

vs.	**NLP4J**	**MaltParser**	**MSTParser**
NLP4J	-	72.35%	71.27%
MaltParser	74.89%	-	76.89%
MSTParser	72.26%	74.84%	-

Table 3: Distribution of *prep*-label *UAS*

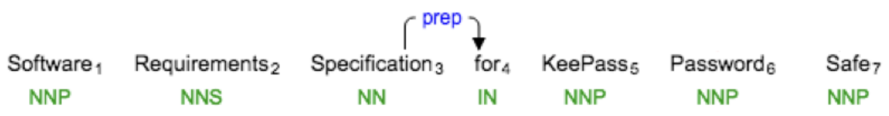

Figure 1: Example of a consistent parsing

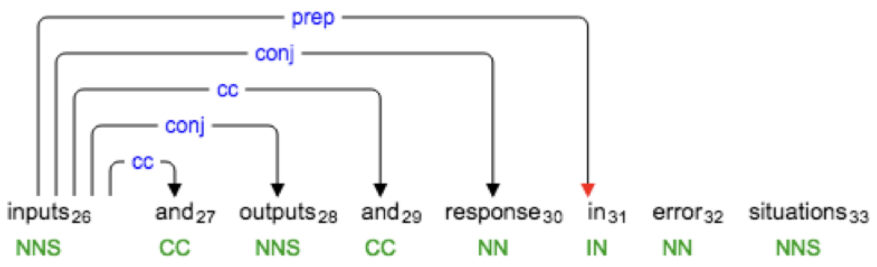

Figure 2: Ambiguous phrase parsed by MaltParser

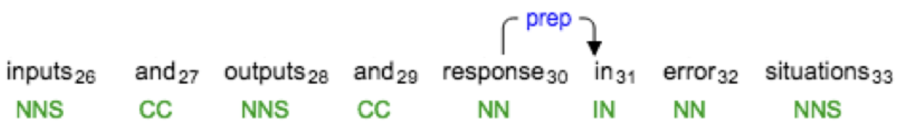

Figure 3: Ambiguous phrase parsed by MSTParser

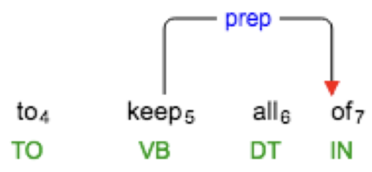

Figure 4: Parsing error by NLP4J parser

RQ2: To which extent are parsing differences caused by ambiguity?

In a manual examination of the first 25 findings, four are due to ambiguity and 21 are due to parsing errors.

An example of a detected ambiguity is presented in Figures 2 and 3. Two interpretations are possible for this phrase. Either the error situations apply to all inputs, outputs, and responses, as understood by MaltParser. Or the error situation is only applied to the response, as determined by MSTParser.

We also present an example of a parsing error by the NLP4J parser in Figure 4.

4.5 Discussion

In this section we discuss the results obtained in Section 4.4.

RQ1 results in an *UAS* ranging from 69.64% to 76.89%. An example of these consistently parsed prepositions is presented in Figure 1 which shows a very simple prepositional construction.

The upper bound of the *UAS* is limited by the accuracy of the dependency parser. In Section 3.2 an example is presented where the NLP4J parser reached 92.26% accuracy. If we were to assume that all three parsers in our prototype could reach that accuracy and were we to assume that findings occur randomly, then we could expect an UAS of $(92.26\%)^3$, that is 78.53%. Of course both assumptios are not true, but this number can aid as context to our *UAS*. In conclusion of RQ1, we find that our prototype is accurate enough for its findings to be used in the evaluation of RQ2.

RQ2 shows that four of 25 findings, that is 16%, were caused by ambiguity. We conclude that the approach of comparing dependency parsers can detect ambiguities.

In summary, our implementation fulfills the goal of Section 4.1. It is accurate enough to produce findings to be evaluated and some of these findings detected are real ambiguities in the study object.

5 Learnings and Future Work

The goal of this report was to evaluate to which extent ambiguities in SRS can be detected by comparison of dependency parsers. We will first summarize the learnings of this report. Afterwards, we present future work.

5.1 Learnings

The learnings of this report consist of three main results.

First, in Chapter 3, we learned how to implement a natural language processing pipeline using *DKPro Core* and how to implement three different dependency parsers. We furthermore learned about issues during the selection of dependency parsers presented in Section 3.2 where we had to compare different sets of dependency labels.

Secondly, we learned that our prototype was of accurate enough for our evaluation, as described in the case study of Chapter 4. For this purpose we calculated the *Unlabeled Attachment Score (UAS)* of *IN*-tags and *prep*-labels and found them to range from 69.64% to 76.89%. The remaining roughly 25% to 30% that are not consistent between parsers were findings to be considered further.

Thirdly, we learned that ambiguities can be detected from these findings. We evaluated the first 25 findings and found four of these to be caused by ambiguity.

In summary, we conclude that ambiguity detection is possible by comparison of dependency parsers. We imagine that similar to how text-editing programs have spell-checking tools, one can imagine an "ambiguity-checking" tool. Of course such a tool would need to be more accurate and it could only check for precision and not for recall. A human writing a SRS would still have to evaluate ambiguities manually, but an "ambiguity-checking" tool could be of great help.

5.2 Future Work

This section presents possible threats to the validity of this report as the scope of this report was rather small.

First of all, the quality of the implementation in Chapter 3 and therefore the accuracy of our dependency parsers for the evaluation in Chapter 4 is entirely dependent on the parsers themselves. Choi et al. (2015) compared ten different dependency parsers [5] while we only selected three, mainly because of their dependency label output. We also did not optimize the parsers any further than their default settings.

Secondly, the amount of ambiguities that can be found is dependent on the study object. We described in Section 4.2 why we selected our *KeePass* study object, but we could have chosen any of the others prepared by Ferrari et al. (2017) [14].

Lastly, we were the only rater in our subjective evaluation of whether an ambiguity exists or not. Ideally multiple researchers would also assess this, separate from each other. We could then calculate Cohen's kappa coefficient [6] to measure the inter-rater agreement between these raters. If ratings were inconsistent, then differences could have been discussed and an agreement on whether a finding was a ambiguity or not could have been reached. Unfortunately this was also out of scope.

All in all, these threats are also starting points for future work.

Bibliography

[1] V. R. Basili, G. Caldiera, and H. D. Rombach. "The Goal Question Metric Approach." In: *Encyclopedia of Software Engineering*. Wiley, 1994.

[2] D. M. Berry, E. Kamsties, and M. M. Krieger. "From contract drafting to software specification: Linguistic sources of ambiguity, 2003." In: *A Handbook*. 2003.

[3] B. W. Boehm, R. K. McClean, and D. B. Urfig. "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software." In: *IEEE Transactions on Software Engineering* 1.1 (Mar. 1975), pp. 125–133.

[4] J. D. Choi. "Dynamic feature induction: The last gist to the state-of-the-art." In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2016, pp. 271–281.

[5] J. D. Choi, J. Tetreault, and A. Stent. "It depends: Dependency parser comparison using a web-based evaluation tool." In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Vol. 1. 2015, pp. 387–396.

[6] J. Cohen. "A Coefficient of Agreement for Nominal Scales." In: *Educational and Psychological Measurement* 20.1 (1960), pp. 37–46.

[7] M.-C. De Marneffe and C. D. Manning. *Stanford typed dependencies manual*. Tech. rep. Stanford University, 2008.

[8] *DependAble - Web-based Visualization and Evaluation Tool for Dependency Parsing*. URL: `https://github.com/emorynlp/dependable` (visited on 06/22/2018).

[9] *DKPro Core Component Reference - Nlp4JPosTagger*. URL: `https://dkpro.github.io/dkpro-core/releases/1.9.2/docs/component-reference.html#engine-Nlp4JPosTagger` (visited on 06/22/2018).

[10] *DKPro Core Component Reference - Nlp4JSegmenter*. URL: `https://dkpro.github.io/dkpro-core/releases/1.9.2/docs/component-reference.html#engine-Nlp4JSegmenter` (visited on 06/22/2018).

[11] *DKPro Core Format Reference - XMLTextReader*. URL: `https://dkpro.github.io/dkpro-core/releases/1.9.2/docs/format-reference.html#format-de.tudarmstadt.ukp.dkpro.core.io.xml.XmlTextReader` (visited on 06/22/2018).

[12] *DKPro Core - Welcome*. URL: `https://dkpro.github.io/dkpro-core/` (visited on 06/22/2018).

[13] R. Eckart de Castilho and I. Gurevych. "A broad-coverage collection of portable NLP components for building shareable analysis pipelines." In: *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT*. Dublin, Ireland: Association for Computational Linguistics and Dublin City University, Aug. 2014, pp. 1–11.

[14] A. Ferrari, G. O. Spagnolo, and S. Gnesi. "PURE: A Dataset of Public Requirements Documents." In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*. Sept. 2017, pp. 502–505.

[15] R. L. Glass. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Upper Saddle River, NJ, USA: Prentice Hall, 1997.

[16] J. E. Kasser, W. Scott, X.-L. Tran, and S. Nesterov. "A Proposed Research Programme for Determining a Metric for a Good Requirement." In: *The Conference on Systems Engineering Research*. CiteseerX, 2006.

[17] *MaltParser - Overview*. URL: `http://www.maltparser.org/` (visited on 06/22/2018).

[18] *MaltParser - Pre-trained models*. URL: `http://www.maltparser.org/mco/mco.html` (visited on 06/22/2018).

[19] *MSTParser*. URL: `https://sourceforge.net/projects/mstparser/` (visited on 06/22/2018).

[20] *NLP4J - Dependency Parsing*. URL: `https://emorynlp.github.io/nlp4j/components/dependency-parsing.html` (visited on 06/22/2018).

[21] *NLP4J - NLP Toolkit for JVM Languages*. URL: `https://emorynlp.github.io/nlp4j/` (visited on 06/22/2018).

[22] *NLP4J - Part-of-Speech Tagging*. URL: `https://emorynlp.github.io/nlp4j/components/part-of-speech-tagging.html` (visited on 06/22/2018).

[23] *PURE - Natural Language Requirements Dataset*. URL: `http://fmt.isti.cnr.it/nlreqdataset/` (visited on 06/22/2018).