2017

Aspect Identification and Sentiment Analysis in Text-Based Reviews

Sean Byrne
Lehigh University

Aspect Identification and Sentiment Analysis in Text-Based Reviews

by

Sean Byrne

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Industrial & Systems Engineering

Lehigh University

May 2017

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

Date

Martin Takáč, Thesis Advisor

Tamás Terlaky, Chairperson of Department

Acknowledgements

I would like to express my gratitude to Professor Martin Takáč for his guidance and encouragement throughout my research and my time at Lehigh. I'd also like to thank my parents, Kevin and Jenifer Byrne, and my brothers Matthew and Jake for their constant support in everything I do. In addition, I'd like to thank Dr. Ali Yazdanyar, physician at Reading Hospital, for allowing me to apply what I've learned in a real-world setting.

Contents

List of Figures

List of Tables

Abstract

Online text-based reviews are often associated with only an aggregate numeric rating that does not account for nuances in the sentiment towards specific aspects of the review's subject. This thesis explores the problem of determining review scores for specific aspects of a review's subject. Specifically, we examine two important subtasks - aspect identification (identifying specific words and phrases that refer to aspects of the review subject) and aspect-based sentiment analysis (determining the sentiment of each aspect). We examine two different models, conditional random fields and an association mining algorithm, for performing aspect identification. We also develop a method for performing aspect-based sentiment analysis based on VADER, a sentence-level sentiment analysis algorithm built for sentiment analysis of social media. We identify key problem considerations, including other important subtasks and ideal training dataset qualities, for future development of a full aspect-based review system.

Chapter 1

Introduction

1.1 Automatic Aspect-Based Review System

Text-based reviews found online have become a common way to evaluate options when making a decision. These reviews span subjects from a variety of topics - products available for purchase online, downloadable applications, movie and music releases, restaurants, hotels, and more. Oftentimes, these reviews are associated with an overall numeric rating (typically on a 5-point or 10-point scale), which can be aggregated to form an average rating for a given subject. However, these ratings oftentimes hide the details present in the text of the reviews. For example, by examining a set of laptop reviews with an average rating of 3.0 out of 5.0, one might find that the screen of the laptop is mostly referred to positively, but the keyboard is mostly referred to negatively. This nuance is not reflected with an overall 5-point numeric rating, despite the fact that users oftentimes have preferences that require a more detailed view of the subject.

In order to more accurately reflect how reviewers feel about different aspects of a subject, it is desirable to develop a system to rate the major features of a subject separately, providing more meaningful information to those who may have specific preferences. A shopper looking to purchase a laptop, for example, may desire a high screen quality while not caring much about the processing power. This shopper would benefit from finding a laptop with a highly-rated score for the aspect "Screen" and may not mind if the laptop's overall score is dragged down by a lower rating for the aspect "Processing Power". It's possible that websites aiming to have a more comprehensive set of ratings could force users to rate specific qualities on a numeric scale, rather than just the overall product. However, this requires more effort on the end user, and ignores the vast amount of text-based review data that already exists.

One way such a system can be developed using existing product reviews is to utilize *sentiment analysis* (also known as opinion mining). Sentiment analysis attempts to derive measures of subjectivity from written text, typically labeling text using either the labels "subjective" and "objective" (ignoring polarity of subjective text), or the labels "positive", "negative", and "neutral" (where "positive" and "negative" are opposite categories of subjectivity, and "neutral" is equivalent to "objective"). Text-based reviews are an important source of data for sentiment analysis because they consist primarily of subjective opinions, making them particularly useful for building models with the ability to determine sentiment polarity.

However, rather than attempting to determine the sentiment of the review as a whole, the sentiment of particular attributes of the product would be measured. If a particular attribute is found to be associated with positive or negative polarity for most instances within a set of reviews, then it is given a high or low rating, respectively, for that particular attribute. These attributes (or aspects) can be found through *aspect identification* - determining what words and phrases (terms) refer to specific aspects of the subject. For example, in the sentence "The battery life is quite strong and lasts all day long," the phrase "battery life" is an aspect term of the subject.

Once these aspect terms have been identified, sentiment analysis can be used to determine the sentiment polarity of each aspect term. Specifically, *aspect-based sentiment analysis* attempts to determine the sentiment of each aspect term. Accurately determining the polarity of aspect terms is more challenging than the typical sentiment analysis task. Sentiment analysis relies heavily on sentiment lexicons that classify adjectives based on their sentiment polarity, but an adjective that has a positive sentiment when used to describe one aspect may have a negative or neutral sentiment when used to describe another aspect. For example, “long” tends to have a positive sentiment when used to refer to “battery life” in a laptop, but a negative sentiment when used to refer to “wait times” at a restaurant. Another significant issue is when multiple aspect terms are mentioned within the same sentence. If one aspect has a positive sentiment and another has a negative sentiment, determining these sentiments accurately requires understanding which portions of the sentence apply to a given aspect term.

In the remainder of Chapter 1, we examine a brief background of natural language processing and mention an ongoing application of the methods describes in this thesis. In Chapter 2, we describe the datasets used and qualities of a useful dataset for the problems of aspect term extraction and aspect-based sentiment analysis, as well as important features that can be derived from the text. In Chapter 3, we examine methods for extracting aspect terms from these datasets, and in Chapter 4, we examine methods for determining the sentiment of aspect terms.

1.2 Natural Language Processing

Natural Language Processing (NLP) is a field of study within computer science and artificial intelligence that focuses on analyzing and deriving meaningful information from human (natural) language. Natural language processing developed as a result of interest in machine translation (MT), the problem of translating sentences automatically from one language to another, in the 1950s. Research was severely limited due to the relatively undeveloped state of computers at the time. Initial research started as dictionary-based, with attempts to translate sentences word-for-word, but issues with determining the correct syntax (the arrangement of words) and semantics (the meaning of words) in translation quickly showed the limitations of such an approach. Despite technological limitations, research of this time period was able to effectively identify the importance of developing an explicit structure and definition for language that could allow methods to be generalized and implemented with computers [16]. The low quality of the methods developed, however, led a committee commissioned by the United States government called ALPAC (Automatic Language Processing Advisory Committee) to express doubts in the merit of continued MT research in a report in 1966 [8]. The committee suggested that significant improvements in computational linguistics was needed before MT could be effectively tackled, leading to a significant shift away from MT in the late 1960s. This shift allowed other problems within NLP to be explored, eventually leading to the broad range of problems studied within the field today.

The massive amount of data and processing power that are accessible today has opened the door to new heights in the world of natural language processing. Modern NLP research examines problems such as converting speech to text [12], answering text-based questions [13], automatically summarizing large documents, automatic spell-checking, determining grammatical relationships between words, and much more. NLP has been utilized in a large variety of business contexts as well. Lawyers use NLP software to analyze large sets of legal documents to find meaningful information. Spam filters utilize NLP to find patterns within email text that indicate a high likelihood of being spam, and Google uses NLP in their language-translation software. Various social media sites utilize natural language processing so that advertisements can be customized to the interests of each user.

In this thesis, we utilize some commonly-used software for natural language processing. In particular, we make extensive use of the Natural Language Toolkit (NLTK) [6] and Stanford's CoreNLP toolkit [18]. These are packages for Python that provides a large set of functions and datasets useful for natural language processing.

1.3 Application: Reading Hospital

With the rise of electronic medical records, applications have started to appear within the medical field. Taking text-based data from the past (in this case, from physicians' notes) and data related to the eventual treatment of the patient's medical issues (for example, procedures done, diagnoses given, and success/failure rates), patterns can be found within the text of the doctors' notes. In this way, physicians' notes can be analyzed to determine signs of postsurgical complications, or to determine the procedure with the highest likelihood of success for a given diagnosis and set of physical traits.

One potential area for the application of natural language processing techniques is a project recently started at Reading Hospital. Because of the nature of this work, the specific details cannot be shared in this paper. However, a basic problem outline can be shared. When a scan is done to examine a particular portion of the body, secondary information can be gathered. For example, in a CT scan where the primary objective is to examine a tumor, secondary nodules could be found on the scan that aren't directly related to the tumor. In this case, the doctor often suggests that the patient make a follow-up appointment with another practitioner; however, there is no easy way to connect the patient to the appropriate office for a follow-up appointment, and oftentimes patients end up ignoring these secondary findings until their next appointment months or even years later. Complications that could have been treated easily, if dealt with earlier, can end up becoming much more serious medical issues because of this.

The project's goal is to use NLP techniques to identify keywords in the notes of these scans that suggest a secondary finding should be examined or a follow-up appointment is needed. The details of these patients and scans could then be routed to the appropriate place automatically. In this way, the methods discussed in this paper can have a real impact on the patients at Reading Hospital.

Chapter 2

Dataset Structure and Text Features

2.1 Datasets

There is a great deal of text-based review data available online - however, the raw data alone isn't enough. In order to perform the three major tasks associated with aspect-based sentiment analysis, the data provided must contain information about which words are aspect terms, which words are a part of which aspect categories, and whether each instance of a term is referred to positively or negatively. This requires human tagging of datasets, along with cross-validation measures to ensure that the tags are consistent across multiple people.

The difficulty of creating adequate data sources causes significant issues when tackling the problem of aspect-based sentiment analysis. It significantly limits the effectiveness of methods that rely heavily on domain-based features, since each subject (spanning all categories of online products, media, restaurants, and others) may require a different set of training data for these methods to be effective. Thus, the importance of developing a model that is not overly reliant on the domain of the training data is particularly important.

SemEval (also known as the International Workshop on Semantic Evaluation) is "an ongoing series of evaluations of computational semantic analysis systems" hosted annually by SIGLEX (Special Interest Group on the Lexicon of the Association for Computational Linguistics) [1]. Each year, a set of tasks related to semantics within natural language processing are developed, with the goals of developing methods of discerning meaning from language and identifying issues worth exploring further. From 2014 to 2016, one of the tasks was "Aspect-Based Sentiment Analysis" [5] [4] [24]. In this task, participants were given data annotated with aspect terms, aspect categories, and their polarities. The goal of the task was to predict each of these for a set of testing data as accurately as possible.

We utilize datasets from the 2014-2016 SemEval competitions. They have been cross-validated to ensure that inter-annotator agreement is high [23], and there is data available from two different domains: laptop and restaurant reviews. The sentences in each years' data are largely the same, but the format they're stored in (as well as their aspect term and aspect category annotations) vary. In all formats, aspect terms and/or aspect categories are associated with a sentiment polarity from the set {"positive", "negative", "neutral"}, though the 2014 and 2016 formats also allow for a fourth "conflict" value that represents subjective statements without clear overall positive or negative sentiment.

```xml
<sentence id="2846">
    <text>Not only was the food outstanding, but the little 'perks' were great.</text>
    <aspectTerms>
        <aspectTerm term="food" polarity="positive" from="17" to="21"/>
        <aspectTerm term="perks" polarity="positive" from="51" to="56"/>
    </aspectTerms>
    <aspectCategories>
        <aspectCategory category="food" polarity="positive"/>
        <aspectCategory category="service" polarity="positive"/>
    </aspectCategories>
</sentence>
```


FIGURE 2.1: An example of the dataset format in SemEval 2014.

The 2014 datasets are stored as sentences (without review context) in two different domains: laptop reviews with 3,141 sentences and restaurant reviews with 3,145 sentences. Aspect terms are provided for sentences in the datasets of both domains, and aspect categories are provided for sentences in the dataset of the restaurant domain. The sentiment polarity fields in this dataset support the ”conflict” value when the dominant sentiment polarity is not clear. For each aspect term, character offsets are provided (in two fields: ”from” and ”to”, representing the beginning and end of the term, respectively) to identify the location of each aspect term within the sentence. Offsets start at index 0 within a sentence, and the ”to” field stores the index of the offset immediately after the last character of the aspect term. Each sentence contains zero or more aspect terms and zero or more aspect categories.

The 2015 datasets are stored as reviews in two different domains: laptop reviews and restaurant reviews. Each review is provided as a list of sentences in order, and each sentence is associated with zero or more aspect categories. Aspect categories are stored as pairs of entities (E) and attributes (A) in the following format: ”E#A”. Entities are components of the overall topic - for example, entities in the set of Laptop reviews include ”CPU”, ”Software”, ”Shipping”, and ”Support”. Attributes are specific features or qualities of the entities - for example, attributes in the set of laptop reviews include ”Price”, ”Quality”, and ”Portability”. This dataset does not support the ”conflict” value

```
<Review rid="720418">
    <sentences>
        <sentence id="720418:0">
            <text>Great Indian food and the service is incredible.</text>
            <Opinions>
                <Opinion target="Indian food" category="FOOD#QUALITY" polarity="positive" from="6" to="17"/>
                <Opinion target="service" category="SERVICE#GENERAL" polarity="positive" from="26" to="33"/>
            </Opinions>
        </sentence>
        <sentence id="720418:1">
            <text>The owner truly caters to all your needs.</text>
            <Opinions>
                <Opinion target="owner" category="SERVICE#GENERAL" polarity="positive" from="4" to="9"/>
            </Opinions>
        </sentence>
        <sentence id="720418:2">
            <text>When family came in he gave them apps to test their palets, and then ordered for them.</
                text>
            <Opinions>
                <Opinion target="NULL" category="SERVICE#GENERAL" polarity="positive" from="0" to="0"/>
            </Opinions>
        </sentence>
        <sentence id="720418:3">
            <text>Everyone was more then happy with his choices.</text>
            <Opinions>
                <Opinion target="NULL" category="SERVICE#GENERAL" polarity="positive" from="0" to="0"/>
            </Opinions>
        </sentence>
        <sentence id="720418:4">
            <text>Great food and the prices are very reasonable.</text>
            <Opinions>
                <Opinion target="food" category="FOOD#QUALITY" polarity="positive" from="6" to="10"/>
                <Opinion target="NULL" category="RESTAURANT#PRICES" polarity="positive" from="0" to="0"/>
            </Opinions>
        </sentence>
    </sentences>
</Review>
```

FIGURE 2.2: An example of the dataset format in SemEval 2015.

for sentiment polarity. For reviews in the Restaurant dataset, an opinion "target" can be specified - this happens when an entity E is explicitly referenced through a target word or phrase in the sentence. This allows aspect terms to be linked to aspect categories. The keyword "NULL" is used if an aspect category's entity is not explicitly referenced through a target. If an opinion target is specified, "from" and "to" fields are used to specify the location of the target within the sentence. These are set to 0 when the target is "NULL".

The 2016 dataset is provided in two different formats. One is identical to the 2015 dataset format. The other is a review-based format that stores sentences and aspect categories separately. Each review consists of a list of sentences and a separate list of the aspect categories within the review. This means that polarity ratings for each aspect category are review-level rather than sentence-level, and so each aspect category is assigned the sentiment polarity that is dominant within most sentences that contain

```xml
<Review rid="1032695">
    <sentences>
        <sentence id="1032695:0">
            <text>Every time in New York I make it a point to visit Restaurant Saul on Smith Street.</text>
        </sentence>
        <sentence id="1032695:1">
            <text>Everything is always cooked to perfection, the service is excellent, the decor cool and
                understated.</text>
        </sentence>
        <sentence id="1032695:2">
            <text>I had the duck breast special on my last visit and it was incredible.</text>
        </sentence>
        <sentence id="1032695:3">
            <text>Can't wait wait for my next visit.</text>
        </sentence>
    </sentences>
    <Opinions>
        <Opinion category="RESTAURANT#GENERAL" polarity="positive"/>
        <Opinion category="FOOD#QUALITY" polarity="positive"/>
        <Opinion category="SERVICE#GENERAL" polarity="positive"/>
        <Opinion category="AMBIENCE#GENERAL" polarity="positive"/>
    </Opinions>
</Review>
```


FIGURE 2.3: An example of the dataset format in SemEval 2016.

the aspect category. Aspect categories are defined in a similar way to the 2015 dataset, using an entity-attribute pair to represent each category. In cases where the dominant sentiment polarity is not clear, the polarity is defined as "conflict". Opinion targets are not provided.

The formats can be summarized as follows. The 2014 dataset identifies specific aspect terms and their associated polarities, as well as aspect categories for the Restaurant dataset that are not explicitly linked to aspect terms. The 2015 dataset is somewhat more specific - it identifies specific entity-attribute combinations that form aspect categories, as well as target aspect terms for the Restaurant dataset that explicitly link aspect terms to aspect categories. It also provides context information by grouping sentences by each review. The 2016 dataset is more general - it identifies specific entity-attribute combinations that form aspect categories that are found within a review as a whole, rather than individual sentences. By comparing methods across dataset formats with very similar data, the value of creating training datasets with a higher or lower level of detail can be found.

2.2 Text Features

2.2.1 Token-level Features

We break each sentence down into tokens consisting of words and punctuation using the Penn Treebank tokenizer within NLTK [20]. This tokenizer splits contractions (for example, “don’t” will become the separate tokens “do” and “n’t”) and stores punctuation as separate tokens.

Some features can be extracted from individual tokens without the need for information from the rest of the sentence or corpus. We store the original token text, as well as a lowercase version of the token. Several binary features are stored - whether or not the token is punctuation, whether or not it is in ”titlecase” (the first letter of the token is capitalized, and the following letters are all lowercase), and whether the token is a digit. We use a popular word stemmer, PorterStemmer, to store the stem of a given word, removing all prefixes and suffixes from the token [25].

2.2.2 Sentence-level Features

Some features require sentence-level context. The index of each token within the sentence is stored, with 0 being the first token of the sentence. A part-of-speech (POS) tagger using the Penn Treebank tagset is used to tag the part-of-speech for each token in a sentence [20]. The full POS tag and the first 2 characters of the POS tag are stored as separate features, as the first two characters are indicative of a broader category that the following characters are part of (for example, “NN”, “NNP”. “NNS”, and “NNPS” are all tags to describe nouns). Each token also stores information about the previous and next tokens in the sentence - the text, lowercase text, stem, and both POS tag features of the previous and next tokens, storing a default value if the previous or next token doesn't exist.

2.2.3 Review-level Features

Oftentimes, text-based reviews are associated with an overall numeric rating. Our datasets do not have contain numerical rating information, but utilizing these review ratings in an aspect-based sentiment analysis model may yield positive results, and is worth future consideration when designing annotated datasets from online reviews.

2.2.4 Other Possible Features

Many other features are commonly used for natural language processing purposes. WordNet is a lexical database designed to store words based on their word sense (the meaning of the word) rather than the word itself [21]. It contains over 155,000 words and 117,000 synonym sets (sets of words with the same meaning), with over 206,000 word-sense pairs in total [2]. Several other semantic relations are stored as well, such as antonyms. Hypernyms, a semantic "parent" of a given word, and hyponyms, semantic "children" of a given word, are stored - for example, the pair "sport" and "baseball" is a hypernym-hyponym pair. Meronyms and holonyms refer to component parts and the collective whole, respectively - for example, the pair "car" and "wheel" is a holonym-meronym pair. Using WordNet in a natural language processing model, particularly the problems of aspect identification and aspect-based sentiment analysis, would give the model a greater understanding of the relationships between words in a sentence. However, WordNet has been found to not significantly impact the performance of text classification models [19], and the limited tests we performed showed little benefit. Despite this, usage of WordNet in other models for aspect identification and aspect-based sentiment analysis may still be worth exploring.

Word2Vec is a deep learning algorithm that takes sentences as inputs and outputs a vectorization of each distinct word within the training data. This can be used to determine the similarity of one word from another word. Word2Vec also allows for accurate operations among words, meaning syntactic and semantic patterns can be accurately generated. For example, suppose vec(word) is the Word2Vec vector representation of a word. vec('brother') - vec('man') + vec('woman') results in a vector similar to vec('sister'). As a result, relationships among words are encoded in the vectors. Word2Vec was designed for massive datasets, ranging from tens of millions to billions of words, and so attempts to train Word2Vec on the datasets available (with only several thousand sentences available) were unsuccessful. Training Word2Vec on larger datasets available, such as the full English Wikipedia, has resulted in positive results in other aspect identificaton models [23].

Chapter 3

Aspect Identification

3.1 Problem Description

In some texts, particularly text-based reviews, there is an overall subject being discussed throughout the text. Aspect identification (or aspect term extraction) is the process of identifying what words and phrases (terms) refer to specific aspects of a subject in these texts.

Aspect identification typically refers to extracting aspect terms explicitly mentioned within the sentence, rather than implied terms. For example, the sentence "The restaurant was quite expensive" does not explicitly mention price, but "expensive" is an adjective referring to the price of the food, an implicit aspect within the sentence. We consider only explicit aspect terms in this paper.

An ideal system would not rely heavily on the domain of the training data, as otherwise a new set of training data would be required for each new domain examined. Identifying aspect terms requires human identifiers to manually record these aspect terms and their sentiment, and requires a consistent approach so that these human identifiers mostly agree with each other. When each set of training data requires potentially hundreds of reviews (thousands of sentences), this task becomes infeasible to complete for the many domains available for text-based reviews.

One of the most significant challenges in aspect identification is balancing accuracy with robustness. The most accurate models will likely require more detailed training data - accurate sentence-level datasets identifying aspect terms and their respective polarities (positive, negative, or neutral). But the most domain-neutral models will rely on more general features and potentially unsupervised methods. Thus, we examine both supervised and unsupervised approaches, and test across domains to see how applicable each supervised method is to training data from a different domain.

3.2 Sequential Labeling: Conditional Random Fields

Aspect term extraction can be modeled as a sequence labeling problem, where each sentence is examined as a sequence of tokens, taking the context of an individual token into account. This framework is used for problems such as part-of-speech tagging, named entity recognition, and shallow parsing [26]. We describe and implement a common sequence labeling model called a Conditional Random Field (CRF), a generalization of another model called a Hidden Markov Model. These are sequential labeling models based on generalizations of the single-label models described with the naive Bayes classifier and Maximum Entropy models. The goal of a CRF is to determine the conditional distribution of potential labels (in our case, using the IOB2 tagging format) given the output (each token's text). Using the framework for Maximum Entropy models and

CRFs, feature functions can be defined that can allow a vector of output features to be associated with each word in a sentence.

3.2.1 Labeling Method

We use the IOB2 tagging format, where each token is associated with one of three labels - inside an aspect term (”I”), outside an aspect term (”O”), or the beginning of an aspect term (”B”). All aspect terms start with a ”B”, so only multi-token aspect terms utilize the ”O” label.

3.2.2 Background: Naive Bayes and Maximum Entropy Models

The naive Bayes classifier is used to predict a class label y given a feature vector $\mathbf{x}$. It is based on the assumption of conditional independence of the individual features given the class label. The model attempts to maximize the joint probability $p(\mathbf{x}, y)$ of the features and the class label, which due to their conditional independence can be described as follows:

$$p(\mathbf{x}, y) = p(y) \prod_{i=1}^{n} p(x_i \mid y). \tag{3.1}$$

The Maximum Entropy classifier (also known as multinomial logistic regression) makes the assumption that $\log(p(y \mid \mathbf{x}))$ can be represented as a linear combination of the features and a constant. This is useful in that the features are not assumed to be independent, and so the relationships among the output features are considered. The Maximum Entropy classifier models the conditional probability $p(y \mid \mathbf{x})$ as follows:

$$p(y \mid \mathbf{x}) = \frac{1}{Z} \exp(\boldsymbol{\beta_y} \mathbf{x} + \beta_{y,0}). \tag{3.2}$$

$Z = \sum_y \exp(\boldsymbol{\beta_y} \mathbf{x} + \beta_{y,0})$ is a normalization constant which adjusts to ensure valid probabilities. The parameters $\boldsymbol{\beta_y}$ and $\beta_{y,0}$ can be chosen based on the training data using the expectation-maximization (EM) algorithm [11].

Naive Bayes is a generative model, meaning that the model estimates the joint probability distribution of the state and the feature vector and uses this learned distribution to predict the likelihood of a feature vector $\mathbf{x}$ being assigned a class label y. Maximum Entropy models, on the other hand, are discriminative - they learn the conditional probability $p(\mathbf{y} \mid \mathbf{x})$ of being in a state $\mathbf{x}$ given an output $\mathbf{y}$. This is important because unlike generative models, the probability distribution of outputs $p(x)$ does not need to be learned. In the case of natural language processing where the observed outputs are words, there are almost certainly words that don't exist in the training corpus that may occur when using the model, meaning $p(x)$ cannot be accurately estimated without training data that contains every possible word - an unfeasible task.

Because these classifiers only predict a single class label for a set of features, they cannot model the relationships among the hidden states. Graphical models such as Hidden Markov Models and CRFs, on the other hand, are able to account for the dependencies between the nodes' labels.

3.2.3 Hidden Markov Models

One model for labeling sequences of inputs is called a Hidden Markov Model (HMM). The system is assumed to be a Markov process, where the state of the current node is dependent only on the state of the previous node in the sequence. However, instead of observing the state of a given node directly, we observe an output that is dependent on the state, and each state has a probability distribution over the set of outputs. HMMs also assume conditional independence of the output features given each node's state, making them a generalization of the Naive Bayes classifier. Given a sequence of outputs and information about each state's distribution of possible outputs, we can predict a sequence of hidden states.

In our problem, the sequence of words or tokens within the sentence is the sequence of outputs, and the sequence of labels, using the IOB2 standard, are the hidden states. Our goal is to predict the IOB2 labels of each token within a sentence, using the sentence's tokens as the sequence of output features.

Let $X = (x_1, x_2, ..., x_n)$ be the sequence of observed outputs and $Y = (y_1, y_2, ..., y_n)$ be the sequence of hidden states. x_i can be any value within a set of possible outputs O and y_i can be any value within a set of possible state labels L. We define the transition probability $p(y_i|y_{i-1})$ of the current state given the previous state. The emission probability $p(x_i|y_i)$ is the probability of observing the current output given the state of the node. The joint probability distribution of a sequence of outputs $\mathbf{x}$ and a sequence of hidden states $\mathbf{y}$ can be defined as follows, denoting $p(y_1)$ as $p(y_1 \mid y_0)$ for simplicity:

$$p(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{n} p(y_i \mid y_{i-1}) p(x_i \mid y_i). \tag{3.3}$$

This is a generalization of the joint probability distribution defined in the naive Bayes classifier, and can be rewritten as follows:

$$p(\mathbf{x}, \mathbf{y}) = \exp\left[\sum_{i=1}^{n} log(p(y_i \mid y_{i-1}) + \sum_{i=1}^{n} log(p(x_i \mid y_i))\right]. \tag{3.4}$$

If we by replace $log(p(y_i \mid y_{i-1}))$ with a parameter $\beta_{y_i,y_{i-1}}$, $log(p(x_i \mid y_i))$ with a parameter μ_{x_i,y_i}, and adjust by a normalization factor Z, we can rewrite this further as:

$$p(\mathbf{x}, \mathbf{y}) = \frac{1}{Z}\exp\left[\sum_{i=1}^{n} \beta_{y_i,y_{i-1}} \sum_{i=1}^{n} \mu_{x_i,y_i}\right]. \tag{3.5}$$

These parameters can be indexed based on the set of labels by using indicator functions to determine the appropriate parameter:

$$p(\mathbf{x}, \mathbf{y}) = \frac{1}{Z}\exp\left[\sum_{i=1}^{n} \sum_{j,k \in L} \beta_{j,k} \mathbf{1}_{\{y_i=j\}} \mathbf{1}_{\{y_{i-1}=k\}} \sum_{i=1}^{n} \sum_{o \in O} \sum_{l \in L} \mu_{o,l} \mathbf{1}_{\{x_i=o\}} \mathbf{1}_{\{y_i=l\}}\right]. \tag{3.6}$$

Finally, feature functions can be defined to simplify the notation used. Allow $f_{j,k}(y_i, y_{i-1}, x_i) = \mathbf{1}_{\{y_i=j\}} \mathbf{1}_{\{y_{i-1}=k\}}$ and $f_{o,l}(y_i, y_{i-1}, x_i) = \mathbf{1}_{\{x_i=o\}} \mathbf{1}_{\{y_i=l\}}$ Under this notation, each pair of possible labels (j, k) and each observation-label pair (o, l) has a feature function defined. By indexing these feature functions and their corresponding parameters $\beta_{j,k}$ and $\mu_{o,l}$ using q (with F total functions), we can write the joint probability as follows:

$$p(\mathbf{x}, \mathbf{y}) = \frac{1}{Z}\exp\left[\sum_{i=1}^{n} \sum_{q=1}^{F} \lambda_q f_q(y_i, y_{i-1}, x_i)\right]. \tag{3.7}$$

This notation will allow the differences between HMMs and CRFs to be highlighted.

3.2.4 CRF Model Description

As with HMMs, we define X as the sequence of hidden states and Y as the sequence of outputs. However, unlike HMMs, feature functions can be defined that can account for other output features. In the case of aspect term extraction, this means that the token features defined in the previous chapter can be used to train the model. [27]

Consider the joint probability distribution for HMMs. The conditional probability $p(\mathbf{y} \mid \mathbf{x})$, derived from the joint distribution, is:

$$p(\mathbf{y} \mid \mathbf{x}) = \frac{\exp\left[\sum_{i=1}^{n} \sum_{q=1}^{F} \lambda_q f_q(y_i, y_{i-1}, x_i)\right]}{\sum_{\mathbf{y'}} \exp\left[\sum_{i=1}^{n} \sum_{q=1}^{F} \lambda_q f_q(y'_i, y'_{i-1}, x_i)\right]}. \tag{3.8}$$

This is equivalent to a linear-chain Conditional Random Field with feature functions corresponding to each output. This is a specific sub-case of linear-chain CRFs; more generally, we can describe each output x_i as a vector of features. In our case, this means that rather than using only the word itself as a feature, we can use various features related to the word (such as prefixes/suffixes, part-of-speech tags, or whether capitalization is used). A feature function and corresponding parameter can be defined for any function of the current features, the current label, and the previous label. The general model is described below:

$$p(\mathbf{y} \mid \mathbf{x}) = \frac{\exp\left[\sum_{i=1}^{n} \sum_{q=1}^{F} \lambda_q f_q(y_i, y_{i-1}, \boldsymbol{x_i})\right]}{Z(\boldsymbol{x})}, \tag{3.9}$$

where $Z(\boldsymbol{x}) = \sum_{\mathbf{y'}} \exp\left[\sum_{i=1}^{n} \sum_{q=1}^{F} \lambda_q f_q(y'_i, y'_{i-1}, \boldsymbol{x_i})\right]$ is the normalization constant, computed by summing the feature functions multiplied by their weights over the possible label combinations. The number of possible label combinations becomes very large, but it will be shown that this problem can be averted during training.

3.2.5 CRF Training

Training requires a set of training data $\{(\boldsymbol{x^{(i)}}, \boldsymbol{y^{(i)}})\}_{i=1}^{N}$ consisting of N 'documents' - in our case, sentences. Each sentence has n_i tokens. For sentence i, $\boldsymbol{y^{(i)}} = \{y_1^{(i)}, y_2^{(i)}, \ldots, y_{n_i}^{(i)}\}$ is a sequence of IOB2 labels for a sentence and $\boldsymbol{x^{(i)}} = \{\boldsymbol{x_1^{(i)}}, \boldsymbol{x_2^{(i)}}, \ldots, \boldsymbol{x_{n_i}^{(i)}}\}$ is a sequence of feature vectors, with one feature vector for each token in the sentence. The goal of training is to maximize the conditional log-likelihood for a set of parameters $\theta = \{\lambda_q\}_{q=1}^{F}$.

$$l(\theta) = \sum_{i=1}^{N} \log(p(\boldsymbol{y^{(i)}} \mid \boldsymbol{x^{(i)}})). \tag{3.10}$$

In addition, a technique called *regularization* is often used to smooth the parameters by making a penalty for overfitting:

$$l(\theta) = \sum_{i=1}^{N} \log(p(\boldsymbol{y^{(i)}} \mid \boldsymbol{x^{(i)}})) - \sum_{q=1}^{F} \frac{\lambda_q^2}{2\sigma^2}. \tag{3.11}$$

This assumes a Gaussian prior on the parameters θ, each with a mean of 0 and variance σ^2. The gradient of $l(\theta)$ is:

$$\frac{\partial l}{\partial \lambda_q} = \sum_{i=1}^{N} \sum_{j=1}^{n_i} f_q(y_j^{(i)}, y_{j-1}^{(i)}, \boldsymbol{x_j^{(i)}}) - \sum_{i=1}^{N} \frac{\partial}{\partial \lambda_q} \log(Z(\boldsymbol{x^{(i)}})) - \frac{\lambda_q}{2\sigma^2}. \tag{3.12}$$

where

$$\frac{\partial}{\partial \lambda_q} \log(Z(\boldsymbol{x^{(i)}})) = \sum_{i=1}^{N} \sum_{j=1}^{n_i} \sum_{y,y' \in L} f_q(y, y', \boldsymbol{x_j^{(i)}}) p(y, y' \mid \boldsymbol{x^{(i)}}). \tag{3.13}$$

The partial derivative with respect to λ_q can be interpreted as follows: the first component is the number of observed occurrences of the feature f_q, the second component is the expected number of occurrences of the feature f_q, and the third is the regularization adjustment. At the maximum likelihood solution, the expected and observed occurrences should be equal.

The maximum likelihood function $l(\theta)$ with regularization is strictly concave, and so a global optimum can be found [27]. This can be done with nonlinear optimization algorithms such as L-BFGS, stochastic gradient descent, and others. CRFsuite, a software implementation of conditional random fields, allows various optimization algorithms to be used for this purpose [22].

3.2.6 CRF Evaluation

An important consideration is the method with which ATE systems are evaluated. One key question is whether to apply these methods to distinct aspect terms or to each occurrence of an aspect term. If we evaluate based on distinct aspect terms, then we take the set of predicted distinct aspect terms and compare them to the list of actual distinct aspect terms. However, aspect terms with higher frequency are more valuable, given that our eventual goal is to determine polarity scores for a few most common terms/categories. A model that is able to accurately predict high-frequency aspect terms, but is less effective at predicting low-frequency aspect terms, is more valuable than a model that is better at predicting low-frequency terms than high-frequency terms.

On the other hand, evaluation based on instances of each aspect term can lead to overconfidence in models that can identify some of the most common terms with accuracy, but cannot accurately identify most other terms. Aspect terms with the highest frequencies in the dataset aren't always more important to accurately identify than aspect terms with lower frequencies. An individual aspect term may be more frequent than other aspect terms simply because it has few or no synonyms (for example, "Microsoft Office" has no synonyms, while "price" has several different words representing the same concept).

Thus, we evaluate the methods described in the previous sections with respect to both distinct aspect terms and instances of each aspect term. We use 70% of the data available in each domain for training and 30% for testing. As a review, three of the most common methods of evaluating models are precision, recall, and F-measure. Precision describes the fraction of predicted aspect terms that actually exist in the dataset. Recall is the fraction of true aspect terms that are predicted by the model. F-measure is the harmonic mean of precision and recall.

CRFsuite implements several algorithms to solve for the CRF parameters. Two of the most common optimization algorithms for solving CRFs are provided: L-BFGS and stochastic gradient descent. L-BFGS is a common quasi-Newton method that avoids storing a full approximated Hessian, making is useful for problems such as CRFs where there are often a large number of parameters to be found [17]. Stochastic gradient descent (SGD) is an extension of gradient descent that moves in the direction of a random data point at each iteration. In the CRFsuite implementation, SGD is performed with ℓ^2 regularization to prevent overfitting. Both of these algorithms have been shown to be successful when utilized to solve conditional random fields [28].

TABLE 3.1: The results for CRFs using distinct aspect terms.

Algorithm	**Dataset**	**Precision**	**Recall**	**F-measure**
L-BFGS	Restaurants	0.7003	0.5224	0.5984
SGD	Restaurants	0.6095	0.4187	0.4964
AP	Restaurants	0.6701	0.4004	0.5013
PA	Restaurants	0.6526	0.5346	0.5877
AROW	Restaurants	0.4399	0.5423	0.4859
L-BFGS	Laptops	0.5969	0.3793	0.4639
SGD	Laptops	0.3357	0.3522	0.3438
AP	Laptops	0.5682	0.2463	0.3436
PA	Laptops	0.5935	0.4064	0.4825
AROW	Laptops	0.4349	0.3867	0.4094

Three other algorithms are implemented in CRFsuite as well: Averaged perceptrons (AP), passive aggressive (PA), and Adaptive Regularization of Weight Vectors (AROW). Averaged perceptrons iterates over the training data, updating the feature weights of a perceptron whenever the model cannot make a correct prediction and updating the average feature weights. The final averaged feature weights are returned by the algorithm [7]. Passive-aggressive algorithms define a loss function on predicted instances, aggressively shifting the current parameter estimate when the current training instance has a positive value for the loss function and making no adjustment when the loss function is zero [9].AROW is a variation of confidence-weighted learning, which maintains a Gaussian distribution to measure the confidence in each parameter estimate. It adjusts the model to prevent overly aggressive shifts that can occur when using passive-aggressive updates [10].

The results for distinct aspect terms for both Restaurant and Laptop datasets (using the 2014 format described in Figure 2.1) can be seen in Table 3.1. The best training algorithms for both datasets evaluated with distinct aspect terms were L-BFGS and PA. Overall, CRFs were more effective on the restaurants domain (with a best

TABLE 3.2: The results for CRFs using instances of aspect terms.

Algorithm	**Domain**	**Precision**	**Recall**	**F-measure**
L-BFGS	Restaurants	0.8491	0.7231	0.7810
SGD	Restaurants	0.8036	0.5629	0.6621
AP	Restaurants	0.8246	0.6127	0.7030
PA	Restaurants	0.8182	0.7574	0.7867
AROW	Restaurants	0.6664	0.7430	0.7026
L-BFGS	Laptops	0.8025	0.6119	0.6944
SGD	Laptops	0.4668	0.4268	0.4459
AP	Laptops	0.7460	0.3895	0.5118
PA	Laptops	0.7715	0.6436	0.7018
AROW	Laptops	0.6510	0.6312	0.6409

F-measure of 0.5984 when using L-BFGS) than on the laptop domain (with a best F-measure of 0.4825 when using PA).

The results for instances of aspect terms can be seen in Table 3.2. The best training algorithms for both datasets evaluated with aspect term instances were L-BFGS and PA, with F-measures of 0.7810 and 0.7867 respectively for the Restaurant domain and 0.6944 and 0.7018 respectively for the Laptop domain. Again, the CRF seems to be more effective on the Restaurant domain than the Laptop domain.

The model seems to have significantly higher precision than recall regardless of algorithm and across both distinct and instance-based evaluation methods. This suggests that the models may have difficulty identifying some aspect terms; however, the significant increase in both precision and recall when evaluating the instances of each aspect term suggests that much of this may come from failing to identify infrequent aspect terms.

To see how effective the model would be on data outside of the training domain, we attempt to train each model on one domain and evaluate its performance using testing data from the other domain. These results can be found in Table 3.3 and Table 3.4.

Table 3.3: The results for CRFs using distinct aspect terms across domains.

Algorithm	**Train Domain**	**Test Domain**	**Precision**	**Recall**	**F-Measure**
L-BFGS	Restaurant	Laptop	0.4354	0.1576	0.2315
SGD	Restaurant	Laptop	0.5272	0.0714	0.1258
AP	Restaurant	Laptop	0.3656	0.1675	0.2297
PA	Restaurant	Laptop	0.4084	0.1921	0.2613
AROW	Restaurant	Laptop	0.1635	0.1921	0.1767
L-BFGS	Laptop	Restaurant	0.6176	0.1280	0.2121
SGD	Laptop	Restaurant	0.3918	0.3089	0.3455
AP	Laptop	Restaurant	0.5350	0.1707	0.2589
PA	Laptop	Restaurant	0.5509	0.1870	0.2792
AROW	Laptop	Restaurant	0.3221	0.2134	0.2567

Table 3.4: The results for CRFs using instances of aspect terms across domains.

Algorithm	**Train Domain**	**Test Domain**	**Precision**	**Recall**	**F-Measure**
L-BFGS	Restaurant	Laptop	0.4900	0.1699	0.2523
SGD	Restaurant	Laptop	0.5795	0.0704	0.1256
AP	Restaurant	Laptop	0.4247	0.1754	0.2483
PA	Restaurant	Laptop	0.4935	0.2113	0.2959
AROW	Restaurant	Laptop	0.1909	0.1851	0.1879
L-BFGS	Laptop	Restaurant	0.8216	0.1792	0.2942
SGD	Laptop	Restaurant	0.5581	0.2824	0.3750
AP	Laptop	Restaurant	0.7289	0.1801	0.2888
PA	Laptop	Restaurant	0.7765	0.2516	0.3800
AROW	Laptop	Restaurant	0.5040	0.2308	0.3166

Overall, the quality of the results suffered significantly, suggesting that the model doesn't perform well on data outside of the domain of the training data. However, some of the algorithms used seem to suffer less reduction in quality than others. Using SGD on the Laptops dataset for training, the F-measures for distinct and instances (0.3438 and 0.4459, respectively) for the Laptop testing set are relatively close to their values when using the Restaurant testing set (0.3455 and 0.3750, respectively). Though the overall results were still poor, this suggests that some methods and training datasets may be more generalizable than others. Discovering than area that may worth exploring in the future.

3.3 Association Mining Method (Hu and Liu)

A method based on association mining to find frequent itemsets was defined in [14]. It is a rule-based method that builds a list of candidate itemsets consisting of nouns and noun phrases in each sentence, then prunes them to identify aspect terms. This is based on the notion that reviewers tend to use similar words when describing aspects of a review topic, and so frequently-occurring sets of words are more likely to be aspect terms.

3.3.1 Association Mining Method Description

First, a set of initial candidate itemsets are generated. A list of nouns and noun phrases N, ordered by their placement within the sentence, are extracted from each sentence i as initial itemsets. Pairs and triples of these nouns and noun phrases within each sentence are also considered candidate terms. This is only done for adjacent nouns and noun phrases. More specifically, the extracted pairs and triples can be described as

$$Pairs = \{N_i \cup N_{i+1} : i \in \{1, 2, ..., |N| - 1\}\}$$

$$Triples = \{N_i \cup N_{i+1} \cup N_{i+2} : i \in \{1, 2, ..., |N| - 2\}\}$$

At this point, we have an initial set of candidate terms. We reduce the set of candidate terms down to a set of "frequent" itemsets, as defined by a minimum support level m. This can also be based off a specified percentage of the dataset. All other candidate terms are eliminated.

Two additional pruning measures are taken to reduce the set of candidate terms. An adjusted frequency measure called "p-support" is found that only counts a candidate term in a sentence if it is not a subset of another candidate term within the sentence. For example, if a sentence contained the phrase "ham sandwich" and both "ham" and "ham sandwich" were candidate terms, then this sentence would not count towards the support of "ham" , since it's a subset of another candidate term "ham sandwich" that exists within the sentence. If the p-support of a candidate term is low, and it appears as part of a larger candidate term, the candidate term is likely a component of the larger term. We define a minimum p-support threshold p - if the p-support of a candidate term is less than p and the candidate term is a subset of some other term, we remove it from the set of candidate terms.

Another pruning measure attempts to correct for issues that can arise from using frequent itemsets. When the initial set of candidate terms is created, pairs and triples of nouns and noun phrases are considered candidate terms. However, these words may be relatively far apart within a sentence, suggesting that they might not be part of the same aspect term. For a term within a given sentence, we find the maximum distance between any two adjacent words in the term. Their distance is measured by how many tokens apart they are in the sentence. If this value exceeds a token distance threshold w, then we consider the term non-compact within the sentence. If a term is found to be non-compact in a greater number of sentences than a maximum non-compact frequency threshold c, then the term is discarded.

Algorithm 1 Association Mining Method (Hu and Liu)

Require: List of sentences S, minimum support threshold m, a token distance threshold w, a maximum non-compact frequency threshold c, and a minimum p-support threshold p.

```
T = {}
for sentence ∈ S do
    N = getNounsAndNounPhrases(sentence)
    T = T ∪ {term ∈ N, getAdjacentPairs(N), getAdjacentTriples(N)}
end for
Support = Dictionary() // Default key value is 0
for sentence ∈ S do
    for term ∈ (T ∩ sentence) do
        Support[term] = Support[term] + 1
    end for
end for
T.remove({term : (term ∈ T) and (Support[term] < m)})
P-Support = Dictionary() // Default key value is 0
Non-Compact = Dictionary() // Default key value is 0
for sentence ∈ S do
    for term ∈ (T ∩ sentence) do
        if maxTokenDistance(term, sentence) > w then
            Non-Compact[term] = Non-Compact[term] + 1
        end if
        if (term ⊄ term2) ∀ term2 ∈ ((T − {term}) ∩ sentences) then
            P-Support[term] = P-Support[term] + 1
        end if
    end for
end for
for term ∈ T do
    if (Non-Compact[term] > c) then
        T.remove(term)
    end if
    if ((P-Support[term] < p) then
        for term2 ∈ T − {term} do
            if term.contains(term2) then
                T.remove(term)
            end if
        end for
    end if
end for
return T
```

3.3.2 Association Mining Method Evaluation

An important consideration in aspect term extraction is the idea that aspect terms are likely to be nouns and noun phrases. In the case of one-word aspect terms that are nouns, identification is as simple as finding nouns from each sentence using a part-of-speech tagger, then using other methods to filter out nouns that aren't actually aspect terms. For the case of multi-word aspect terms, noun phrases must be identified. Any unsupervised method for aspect identification must somehow identify these noun phrases without the benefit of training data. The general problem of identifying grammatical structures such as noun phrases is called shallow parsing [3]. Three different methods were explored for identifying noun phrases. We attempted to use NLTK's "Regexp" (regular expression) feature, which finds specific patterns in text using a pre-defined search pattern [6]. However, noun phrases take many possible forms, and defining all the possible search patterns that noun phrases may exist in is unfeasible. We also examined bigram and trigram classifiers, trained on a portion of Treebank data available in NLTK [20]. Finally, we examined the default named-entity chunker within NLTK.

In the testing of the Association Mining algorithm, it became clear that noun chunking was a significant issue that hindered the performance of the algorithm as a whole. After tuning the input parameters for the Restaurant domain dataset, the best model using the named-entity chunker had a precision of 0.3777, recall of 0.2480, and F-measure of 0.2994. This is with a minimum support threshold m of 6, a minimum p-support threshold p of 2, a max token distance threshold w of 2, and a maximum non-compact frequency threshold of 1. We examined the full list of candidate terms before pruning (consisting of all nouns and noun phrases in the sentence, as well as all adjacent pairs and triples of nouns and noun phrases) and found that only 61.18% were detected - this provides an upper bound on the recall of the model. In the future, examining effective ways to identify noun phrases is an important step in improving unsupervised methods, particularly those based on frequent itemsets.

Chapter 4

Aspect-Based Sentiment Analysis

4.1 Problem Description

This chapter is focused on estimating the sentiment of the aspect terms in a sentence, assuming the aspect terms are known. We also examine the case where aspect categories are provided for each sentence rather than individual aspect terms.

Given a set of reviews with aspect terms identified, we would like to accurately estimate the sentiment of each occurrence of an aspect term in a sentence. With one aspect per sentence, an assumption can be made that polarity within the sentence is associated with the polarity of the aspect. When multiple aspect terms are present in one sentence, words associated with one aspect term may incorrectly be associated with another aspect term, causing the polarities of each aspect term within the sentence to affect each other.

An issue with using aspect terms individually is that oftentimes, multiple aspect terms will refer to the same of similar aspects. For example, ”price” and ”cost” refer to the same aspect, yet are considered separate aspect terms. This suggests that a way to categorize aspect terms is desirable when designing a system to accurately rate important aspects of a review's subject. As such, we will focus on accurately identifying the sentiment of instances of aspect terms, rather than the problem of aggregating these terms to provide a more accurate view of a more general "aspect category".

A secondary formulation of the problem can be given for aspect categories (predefined categories that collectively contain the most important or commonly-discussed aspects of a review's subject). Given a set of reviews with sentence-level aspect categories, we would like to accurately estimate the sentiment of each occurrence of an aspect category in a sentence. There are multiple benefits to using aspect categories rather than specific aspect terms. Typically there will be a much smaller number of aspect categories than aspect terms, and these categories will be present in a larger number of sentences than individual aspect terms. This means a smaller amount of data is needed to have enough instances of an aspect category to provide an accurate rating. However, identifying these aspect categories in the first place can be difficult, and requires a predefined list of categories for each domain. As such, the results provided here are predicated on the availability of a method to identify these aspect categories.

4.2 VADER-based Method

VADER, or the Valence-Aware Dictionary for sEntiment Reasoning, is a rule-based model for performing sentiment analysis on a per-sentence basis. The system was trained on online media text, some of which included movie and product reviews. VADER utilizes a sentiment lexicon constructed with the purpose of being generalizable to multiple domains. This makes VADER particularly suitable for analyzing online review data. In addition, by classifying on a per-sentence basis and performing unsupervised, VADER can easily be applied and tested on newly-seen data and data across domains.

Their sentiment lexicon was based on several existing sentiment lexicons, as well as common emoticons and acronyms. It includes valence scores (between -4 and 4) that contain information about sentiment intensity (how strongly a word expresses a sentiment) in addition to sentiment polarity.

Given a sentence, VADER calculates a valence score to measure the sentiment intensity and polarity. Five major heuristics are used to determine the valence score of a given sentence:

1. Some types of punctuation, specifically exclamation points, increases the magnitude of the valence score. For example, "The keyboard is great." is rated with a lower magnitude than "The keyboard is great!"

2. Full-word capitalization, especially when other nearby words aren't fully capitalized, increases the magnitude of the valence score. For example, "The keyboard is great!" is rated with a lower magnitude than "The keyboard is GREAT!"

3. A set of adverbs called 'degree modifiers' is used to increase or decrease the magnitude of the valence score, depending on the word. For example, "The keyboard is great." is rated with a lower magnitude than "The keyboard is very great." and with a higher magnitude than "The keyboard is kinda great."

4. The conjunction "but" signals a shift in sentiment polarity. The sentiment of the portion of the sentence after "but" is considered to be the dominant sentiment, and

contributes a greater amount (two-thirds) to the valence score than the portion of the sentence before "but" (one-third).

5. The trigram before an occurrence of a lexical feature (as determined by the sentiment lexicon) is examined to determine whether a negation is used to express the opposite polarity. For example, "The keyboard is not great" would be given a negative valence score, since "not" is a negation that flips the polarity of "great".

VADER returns a set of four scores: one each for "positive", "negative", and "neutral" (which together sum to 1.0), as well as a "compound" score (ranging from -1.0 to 1.0) reflecting the intensity of the polarity within the sentence. Negative scores are associated with negative polarity within a sentence, and positive score are associated with positive sentence polarity. Larger magnitudes of the "compound" score are associated with higher intensities.

In order to compare these scores with the available data, for each sentence we return a single label ("positive", "negative", or "neutral") depending on the scores returned by VADER. If a sentence's "neutral" score is 1.0, we return the label "neutral". If a sentence's "negative" score is greater than its "positive" score (or if the "compound" score is less than 0), we return "negative". Otherwise, we return "positive". We do not attempt to classify "conflict" values.

4.2.1 Evaluation

We keep track of the predicted and true label values for each occurrence of an aspect term (and additionally for each occurrence of an aspect category, in the case of the restaurant domain dataset). Accuracy is the primary measurement we use to evaluate

TABLE 4.1: The results using VADER on aspect terms in the Laptop domain.

Accuracy:		0.5855	
Label	**Precision**	**Recall**	**F-Measure**
Positive	0.8140	0.6543	0.7254
Negative	0.3362	0.2916	0.3123
Neutral	0.4535	0.7104	0.5536

TABLE 4.2: The results using VADER on aspect terms in the Restaurant domain.

Accuracy:		0.6501	
Label	**Precision**	**Recall**	**F-Measure**
Positive	0.8235	0.7558	0.7882
Negative	0.4028	0.3287	0.3620
Neutral	0.3730	0.6423	0.4719

TABLE 4.3: The results using VADER on aspect categories in the Restaurant domain.

Accuracy:		0.6535	
Label	**Precision**	**Recall**	**F-Measure**
Positive	0.7918	0.7974	0.7946
Negative	0.5226	0.2942	0.3765
Neutral	0.3674	0.6570	0.4713

our model; and Precision, recall, and F-Measure are also calculated with respect to the labels "positive", "negative", and "neutral". The term-based results can be found in Table 4.1 for the laptop domain dataset and in Table 4.2 for the Restaurant domain. The accuracy is reasonably high for an unsupervised model, though it is somewhat lower for the Laptop domain (0.5855) versus the restaurant domain dataset (0.6501). Evaluating based on aspect categories for the restaurant domain dataset provides similar results, without significant variation in any of the evaluation measures. These results can be found in Table 4.3.

4.2.2 Ratings-Based Evaluation

Given the number of positive (p), negative (n), neutral/objective (o), and conflict (c) labels for a given aspect term or aspect category, a rating (r) from 1 to 5 can be

TABLE 4.4: True and predicted ratings for each category in the Restaurant domain.

Category	**True Rating**	**Predicted Rating**	**Rating Error**
food	4.15	4.42	-0.27
ambience	3.81	4.42	-0.61
price	3.44	4.48	-1.04
anecdotes/miscellaneous	3.92	4.15	-0.23
service	3.38	4.05	-0.67

determined as follows:

$$r = 4\left[\frac{p + 0.5c}{p + n + c}\right] + 1. \tag{4.1}$$

This model assumes that "conflict" labels are associated with an equal split between positive and negative sentiment, and assumes that positive occurrences should be weighted the same as negative occurrences. This assumption is based on the idea that a review with n out N stars has a fraction of positive to negative sentiment of $\frac{n-1}{N-1}$. However, this may not be true in practice. Given a dataset with quantitative review scores in addition to annotated aspect terms or categories, more accurate proportions of positive to negative sentiment may be developed. Using these proportions, weights can be used to more heavily skew an occurrence of a particular sentiment label versus other sentiment labels.

We use the restaurant domain dataset's aspect categories to calculate ratings, since there are significantly more occurrences of each aspect category than any one aspect term. The ratings based on the adjusted VADER model's predictions and based on the true sentiment labels can be found in Table 4.4. Overall, the predicted ratings tended to overestimate the true rating by an average of 0.564; this suggests that VADER is somewhat skewed towards positive ratings, at least on our available dataset.

Chapter 5

Conclusion

In this thesis, we explored some of the key tasks in the development of an aspect-based review system. We outlined the considerations required for developing an annotated dataset for the purpose of training models for aspect identification and aspect-based sentiment analysis. For the task of aspect identification, two algorithms were described and tested: a supervised sequential learning model called a conditional random field and an unsupervised association mining algorithm. The results for conditional random fields suggest that they are an effective classifier for identifying aspect terms, particularly when the parameters are learned using L-BFGS or a passive-aggressive algorithm. The results for the association mining algorithm were relatively poor due to issues with identifying noun phrases, but illuminated a future area for further exploration: accurately identifying noun phrases. For the task of aspect-based sentiment analysis, we describe a modified version of VADER, a rule-based sentiment intensity analyzer, to estimate the sentiment of aspect terms and aspect categories. [15]. The results for this model were

A significant area of future exploration is aspect aggregation - identifying aspect terms that are synonyms of each other (for example, "price" and "cost") and aspect terms that are a part of an overarching category (for example, "water" and "wine" might be part of an overarching category called "beverages"). This can be done with predefined categories, which can allow for a supervised approach to the clustering problem. Review-level and sentence-level training data is difficult to generate for a large number of domains, but having a small number of predefined categories to capture the most common aspect terms for each domain is much more feasible. Unsupervised clustering methods may also be explored, given a fixed number of clusters. In this case, clusters can be identified by their most frequent aspects.

Bibliography

[1] SIGLEX (ACL Special Interest Group). `http://alt.qcri.org/siglex/`. Accessed: 2017-04-21.

[2] WNSTATS(7WN) manual page. `http://wordnet.princeton.edu/wordnet/man/wnstats.7WN.html`. Accessed: 2017-04-27.

[3] S. P. Abney. Parsing by chunks. In *Principle-based parsing*, pages 257–278. Springer, 1991.

[4] I. Androutsopoulos, D. Galanis, S. Manandhar, H. Papageorgiou, J. Pavlopoulos, and M. Pontiki. SemEval-2015 Task 12: Aspect Based Sentiment Analysis < SemEval-2015 Task 12. `http://alt.qcri.org/semeval2015/task12/`. Accessed: 2017-04-15.

[5] I. Androutsopoulos, D. Galanis, S. Manandhar, H. Papageorgiou, J. Pavlopoulos, and M. Pontiki. Task Description: Aspect Based Sentiment Analysis (ABSA) < semeval-2014 task 4. `http://alt.qcri.org/semeval2014/task4/`. Accessed: 2017-04-15.

[6] S. Bird. NLTK: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions*, pages 69–72. Association for Computational Linguistics, 2006.

[7] M. Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing-Volume 10*, pages 1–8. Association for Computational Linguistics, 2002.

[8] N. R. Council, A. L. P. A. Committee, et al. *Language and Machines: Computers in Translation and Linguistics; A Report.* National Academy of Sciences, National Research Council, 1966.

[9] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7(Mar):551–585, 2006.

[10] K. Crammer, A. Kulesza, and M. Dredze. Adaptive regularization of weight vectors. In *Advances in Neural Information Processing Systems*, pages 414–422, 2009.

[11] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (methodological)*, pages 1–38, 1977.

[12] J. J. Godfrey, E. C. Holliman, and J. McDaniel. SWITCHBOARD: Telephone speech corpus for research and development. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 517–520. IEEE, 1992.

[13] L. Hirschman and R. Gaizauskas. Natural language question answering: the view from here. *Natural Language Engineering*, 7(04):275–300, 2001.

[14] M. Hu and B. Liu. Mining and summarizing customer reviews. In *Proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 168–177. ACM, 2004.

[15] C. J. Hutto and E. Gilbert. Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Eighth International AAAI Conference on Weblogs and Social Media*, 2014.

[16] K. S. Jones. Natural language processing: a historical review. In *Current Issues in Computational Linguistics: In Honour of Don Walker*, pages 3–16. Springer, 1994.

[17] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1):503–528, 1989.

[18] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL System Demonstrations*, pages 55–60, 2014.

[19] T. N. Mansuy and R. J. Hilderman. Evaluating WordNet Features in Text Classification Models. In *FLAIRS Conference*, pages 568–573, 2006.

[20] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[21] G. A. Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[22] N. Okazaki. CRFsuite: a fast implementation of conditional random fields (CRFs). 2007.

[23] I. Pavlopoulos. Aspect based sentiment analysis. *Athens University of Economics and Business*, 2014.

[24] M. Pontiki, D. Galanis, H. Papageorgiou, S. Manandhar, and I. Androutsopoulos. Task 5: Aspect-Based Sentiment Analysis < SemEval-2016 Task 5. `http://alt.qcri.org/semeval2016/task5/`. Accessed: 2017-04-15.

[25] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[26] F. Sha and F. Pereira. Shallow parsing with conditional random fields. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 134–141. Association for Computational Linguistics, 2003.

[27] C. Sutton and A. McCallum. An introduction to conditional random fields. *arXiv preprint arXiv:1011.4088*, 2010.

[28] S. Vishwanathan, N. N. Schraudolph, M. W. Schmidt, and K. P. Murphy. Accelerated training of conditional random fields with stochastic gradient methods. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 969–976. ACM, 2006.

Appendix - Data Processing and Test Functions

```
import time
from collections import defaultdict
import xml.etree.ElementTree as ET
import libraries.structure as st
from libraries.structure import Corpus
import aspect_identification as ai
import sentiment_analysis as sa
from stanford_corenlp_python import jsonrpc


def sentimentAnalysisTest(data):
    instances = data.corpus
    trueTermPolsBySent = sa.getTermPolarities(instances)
    trueCatPolsBySent = sa.getCategoryPolarities(instances)

    predictedTermPolsBySent = sa.vaderTermPolarities(instances)
    print "Evaluate By Terms:"
    sa.evaluatePolarities(trueTermPolsBySent, predictedTermPolsBySent)

    if len([i for j in trueCatPolsBySent for i in j]) > 0:
        predictedCatPolsBySent = sa.vaderCategoryPolarities(instances)
        print "Evaluate By Categories:"
        sa.evaluatePolarities(trueCatPolsBySent, predictedCatPolsBySent)
        print "True Ratings:"
        print sa.computeRatings(st.fd2([i for j in trueCatPolsBySent for
         ↪  i in j]))
        print "Predicted Ratings:"
        print sa.computeRatings(st.fd2([i for j in
         ↪  predictedCatPolsBySent for i in j]))

def aspectIdentificationTest(dataR, dataL, HL = True, CRF = True):
    # Split into train/test data
    trainR, testR = dataR.split(threshold=0.7)
    trainL, testL = dataL.split(threshold=0.7)
    train = trainR
```

```
    test = testR

    testFD = st.fd([" ".join(a.tokenized_term) for i in test for a in i.aspect_terms])
    testBySent = [i.adjustFormat() for i in test]

    numMethods = 0
    if HL == True:
        # H&L settings
        minSupports = [0]
        minPsupports = [0]
        maxWordDist = [10.0]
        maxNonCompact = [10.0]
        params = [(i,j,k,l) for i in minSupports for j in minPsupports for k in maxWordDist for l in maxNonCompact]
        numMethods += len(params)

    if CRF == True:
        # CRF settings
        algs = ['lbfgs', 'l2sgd', 'ap', 'pa', 'arow']
        numMethods += len(algs)

    predictedFDs = range(numMethods)
    predictedTermsBySent = range(numMethods)
    methodNames = []
    count = 0

    if HL == True:
        for p in range(len(params)):
            # Run Association Mining (Hu & Liu) algorithm
            i = params[p][0]
            j = params[p][1]
            k = params[p][2]
            l = params[p][3]
            predictedFDs[count], predictedTermsBySent[count] = ai.HuLiu(dataL.corpus, minSupport = i, minPsupport = j, maxWordDist = k, maxNonCompact = l)
            methodNames.append("H&L: (minS="+str(i)+",minPS="+str(j)+",maxWD="+str(k)+",maxNC="+str(l)+")"
            count += 1

    if CRF == True:
        for k in algs:
            # Run Conditional Random Field algorithm
            crfLabels = ai.crf(train, test, k)
            predictedFDs[count], predictedTermsBySent[count] = ai.IOB2toAspectTerms(crfLabels, test)

```

```python
                methodNames.append(k)
                count += 1
    # Evaluate methods
    ai.evaluateAspectTerms(testFD, testBySent, predictedFDs, predictedTermsBySent, methodNames, True)


def process_semeval_2015():
    # the train set is composed by train and trial data set
    corpora = dict()
    corpora['laptop'] = dict()
    train_filename = 'datasets/ABSA-SemEval2015/ABSA-15_Restaurants_Train_Final.xml'
    trial_filename = 'datasets/ABSA-SemEval2015/absa-2015_restaurants_trial.xml'

    reviews = ET.parse(train_filename).getroot().findall('Review') + \
        ET.parse(trial_filename).getroot().findall('Review')

    sentences = []
    for r in reviews:
        sentences += r.find('sentences').getchildren()

    # TODO: parser is not loading aspect words and opinioss
    corpus = Corpus(sentences)
    corpus.size()


def process_semeval_2014(type = "R"):
    # the train set is composed by train and trial dataset
    # corpora = dict()
    # corpora['data'] = dict()
    if type == "R":
        train_filename = 'datasets/ABSA-SemEval2014/Restaurants_Train_v2.xml'
        trial_filename = 'datasets/ABSA-SemEval2014/restaurants-trial.xml'

    elif type == "L":
        train_filename = 'datasets/ABSA-SemEval2014/Laptop_Train_v2.xml'
        trial_filename = 'datasets/ABSA-SemEval2014/laptops-trial.xml'
    corpus = Corpus(ET.parse(train_filename).getroot().findall('sentence') +
                    ET.parse(trial_filename).getroot().findall('sentence'))
    # corpora['data']['trainset'] = dict()
    # corpora['data']['trainset']['corpus'] = corpus
    return corpus
```

```python

def main():
    # TODO: start corenlp server "python corenlp.py"

    # interface for Stanford-Core-NLP server
    start = time.time()
    server = jsonrpc.ServerProxy(jsonrpc.JsonRpc20(),
                                 jsonrpc.TransportTcpIp(addr=("127.0.0.1",
                                 8080)))

    #result = loads(server.parse("Hello world.  It is so beautiful"))
    #print "Result", result

    # Load corpus
    dataR = process_semeval_2014("R")
    dataL = process_semeval_2014("L")

    print 'The restaurant corpus has %d sentences, %d aspect term occurrences, and %d distinct aspect terms.' % (dataR.size, sum(dataR.aspect_terms_fd[a] for a in dataR.aspect_terms_fd), len(dataR.top_aspect_terms))
    print 'The laptop corpus has %d sentences, %d aspect term occurrences, and %d distinct aspect terms.' % (dataL.size, sum(dataL.aspect_terms_fd[a] for a in dataL.aspect_terms_fd), len(dataL.top_aspect_terms))

    end = time.time()
    print "Load Corpus: " + str(end - start) + " seconds"
    start = end

    aspectIdentificationTest(dataR, dataL, HL=True, CRF=False)
    sentimentAnalysisTest(dataR)

if __name__ == '__main__':
    main()
```

Appendix - Class Definitions

```
import xml.etree.ElementTree as ET, getopt, logging, sys, random, re, copy
from xml.sax.saxutils import escape
import nltk
from nltk.tokenize import WordPunctTokenizer
from nltk.tokenize import TreebankWordTokenizer as Tokenizer
from nltk.stem.porter import PorterStemmer as Stemmer

import string
from collections import defaultdict

def fd(counts):
    '''Given a list of occurrences (e.g., [1,1,1,2]), return a dictionary of frequencies (e.g., {1:3, 2:1}.)'''
    d = defaultdict(lambda:0)
    for i in counts: d[i] = d[i] + 1 if i in d else 1
    return d

freq_rank = lambda d: sorted(d, key=d.get, reverse=True)
'''Given a map, return ranked the keys based on their values.'''

def fd2(counts):
    '''Given a list of 2-uplets (e.g., [(a,pos), (a,pos), (a,neg), ...]), form a dict of frequencies of specific items (e.g., {a:{pos:2, neg:1}, ...}).'''
    d = {}
    for i in counts:
        # If the first element of the 2-uplet is not in the map, add it.
        if i[0] in d:
            if i[1] in d[i[0]]:
                d[i[0]][i[1]] += 1
            else:
                d[i[0]][i[1]] = 1
        else:
            d[i[0]] = defaultdict(lambda: 0)
            d[i[0]][i[1]] += 1
    return d
```

```

def validate(filename):
    '''Validate an XML file, w.r.t. the format given in the 4th task of **SemEval '14**.'''
    elements = ET.parse(filename).getroot().findall('sentence')
    aspects = []
    for e in elements:
        for eterms in e.findall('aspectTerms'):
            if eterms is not None:
                for a in eterms.findall('aspectTerm'):
                    aspects.append(Aspect('', '').createEl(a).term)
    return elements, aspects


fix = lambda text: escape(text.encode('utf8')).replace('\"','&quot;')
'''Simple fix for writing out text.'''

# Dice coefficient
def dice(t1, t2, stopwords=[]):
    tokenize = lambda t: set([w for w in t.split() if (w not in stopwords)])
    t1, t2 = tokenize(t1), tokenize(t2)
    return 2. * len(t1.intersection(t2)) / (len(t1) + len(t2))

# Find the index of the nth occurrence of a word within a tokenized text
def findNthOccurrence(tokenized_text, word, n):
    if n < 1:
        print "Error: n must be an integer > 1"
        exit()
    k = 0        # How many occurrences we've seen so far
    for index in range(len(tokenized_text)):
        if word in tokenized_text[index]:
            k = k + 1
            if k == n:
                return index
    print "Error: Could not find nth occurrence"
    return -1

def generate(sentences):
    features = [[token.toDict() for token in s.tokens] for s in sentences]
    labels = [[token.actualIOB2 for token in s.tokens] for s in sentences]
    return features, labels

class Category:
```

```
    '''Category objects contain the term and polarity (i.e., pos, neg, neu, conflict) of the category (e.g., food, price, etc.) of a sentence.'''

    def __init__(self, term='', polarity=''):
        self.term = term
        self.polarity = polarity

    def createEl(self, element):
        self.term = element.attrib['category']
        self.polarity = element.attrib['polarity']
        return self

    def update(self, term='', polarity=''):
        self.term = term
        self.polarity = polarity

class Token:
    ''' Token objects contain information about an individual token - usually a word or punctuation. '''

    def __init__(self, text='', index=-1):
        self.text = text                               # The text of the token
        self.index = index                             # Index of the token in the tokenized sentence
        self.isBOS = not index                         # isBOS (Beginning of sentence): True if index = 0, False otherwise
        self.lower_text = text.lower()                 # The lowercase text of the token
        self.isTitle = text.istitle()                  # True if token is "titlecased" (first letter is uppercase and other letters are lowercase)
        self.isPunct = text in string.punctuation      # True if the token is punctuation rather than a word
        self.isDigit = text.isdigit()                  # True if the token is a digit rather than a word
        self.stem = Stemmer().stem(text)               # Word stem of the token (Ex: the stem of "running" is "run")
        self.actualIOB2 = "O"                          # "O" if token is outside, "I" if token is inside, "B" if token is the beginning of an aspect term
        self.polarity = ""                             # Positive ("pos"), negative ("neg"), or neutral ("neu")
        self.POS = ""                                  # Part of speech of the token
        self.POS2 = ""                                 # First 2 characters of the POS tag
```

```python

    def toDict(self):
        features = dict(self.__dict__)
        features.pop('actualIOB2')
        return features

    def setIndex(self, index):
        self.index = index

    def setPrev(self, prev):
        self.prev_text = prev.text
        self.prev_lower_text = prev.lower_text
        self.prev_POS = prev.POS
        self.prev_POS2 = prev.POS2
        self.prev_stem = prev.stem

    def setNext(self, next):
        self.next_text = next.text
        self.next_lower_text = next.lower_text
        self.next_POS = next.POS
        self.next_POS2 = next.POS2
        self.next_stem = next.stem

    def setActualIOB2(self, IOB2):
        self.actualIOB2 = IOB2

    def setPredictedIOB2(self, IOB2):
        self.predictedIOB2 = IOB2

    def setPOS(self, POS):
        self.POS = POS
        self.POS2 = POS[:2]

    def setPolarity(self, polarity):
        self.polarity = polarity


class Aspect:
    ''' Aspect objects contain information about each aspect term. '''

    def __init__(self, term='', id='', tokens=''):
        self.term = term              # The text of the aspect term
        self.id = id                  # The sentence id
        self.offsets = ''             # The offsets within the sentence {'from':startIndex, 'to':endIndex}
        self.polarity = ''            # The polarity (pos, neg, neu, conflict)
```

```
        self.lower_term = ''         # The lowercase text of the aspect term
        self.tokens = ''             # An ordered list of Tokens representing the sentence
        self.tokenized_term = ''     # An ordered list of Strings representing the sentence
        self.termSize = ''           # Number of elements in tokenized_term
        self.headIndex = ''          # The index of the term's first token
        self.endIndex = ''           # The index after the term's last token
        if tokens != '':
            self.createFromTokens(tokens)
        elif len(term) > 0:
            self.lower_term = self.term.lower()
            self.tokenized_term = Tokenizer().tokenize(self.term)
            self.lower_tokenized_term = [t.lower() for t in self.tokenized_term]
            self.termSize = len(self.tokenized_term)

    def createFromTokens(self, tokens):
        ''' Create an Aspect from tokens (used after initial file processing) '''
        self.tokens = tokens
        self.tokenized_term = [t.text for t in tokens]
        self.lower_tokenized_term = [t.lower for t in self.tokenized_term]
        self.termSize = len(tokens)
        self.headIndex = tokens[0].index
        self.endIndex = self.headIndex + self.termSize

    def createEl(self, element):
        ''' Create an Aspect from an XML element (used when reading from file)
            '''
        self.term = element.attrib['term']
        self.lower_term = self.term.lower()
        self.polarity = element.attrib['polarity']
        self.offsets = {'from': str(element.attrib['from']), 'to': str(element.attrib['to'])}
        self.lower_term = self.term.lower()
        self.tokenized_term = Tokenizer().tokenize(self.term)
        self.lower_tokenized_term = [t.lower() for t in self.tokenized_term]
        self.termSize = len(self.tokenized_term)
        return self

```

```python
    def compareWithinSentence(self, otherAspect):
        ''' Comparison based on same sentence - only returns true if the aspects are in the same position within the sentence (ex: the first occurrence of the aspect "keyboard" does not equal the second occurrence of the same aspect term within the sentence)
        '''
        if self.headIndex == otherAspect.headIndex:
            if self.termSize == otherAspect.termSize:
                return True
        return False

    def compare(self, otherAspect):
        ''' Comparison based on the words within the aspect - returns true if all Tokens within the aspect are equivalent.
        '''
        result = False
        if self.termSize == otherAspect.termSize:
            result = True
            for i in range(termSize):
                if self.tokenized_term[i].text != otherAspect.tokenized_term[i].text:
                    result = False
                    break
        return result

    def setTokens(self, tokens):
        self.tokens = tokens

    def getHeadToken():
        return self.tokens[0]

    def setIndices(self, headIndex):
        self.headIndex = headIndex
        self.endIndex = headIndex + self.termSize

    def setOffsets(self, offsets):
        self.offsets = offsets

    def setPolarity(self, polarity):
        self.polarity = polarity

class Instance:
    '''An instance is a sentence, modeled out of XML (pre-specified format, based on the 4th task of SemEval 2014). It contains the text, the aspect terms, and any aspect categories.
    '''

```

```
    def __init__(self, element):
        self.text = element.find('text').text
        self.id = element.get('id')
        self.generateTokens()
        self.aspect_terms = [Aspect('', id=self.id).createEl(e) for es in
                             element.findall('aspectTerms') for e in es if
                             es is not None]
        self.aspect_categories = [Category(term='', polarity='').createEl(e) for es in element.findall('aspectCategories')
                                  for e in es if
                                  es is not None]
        self.updateAspectFields()   # Updates Aspect features related to Tokens, and vice versa

    def generateTokens(self):
        ''' Generate tokens based on the tokenization of the sentence.
            '''

        # Tokenize text and create Token object list
        self.tokenized_text = Tokenizer().tokenize(self.text)
        self.tokens = [Token(self.tokenized_text[i], i) for i in range(len(self.tokenized_text))]

        # Update the POS tag for each Token object
        tagged_text = nltk.pos_tag(self.tokenized_text)
        for i in range(len(self.tokens)):
            self.tokens[i].setPOS(tagged_text[i][1])

        # Update the next and previous tokens for each Token object
        for i in range(len(self.tokens)):
            token = self.tokens[i]
            if i == 0 and i == (len(self.tokens) - 1):
                token.setPrev(Token())
                token.setNext(Token(index=len(self.tokens)))
            elif i == 0:
                token.setPrev(Token())
                token.setNext(self.tokens[i+1])
            elif i == (len(self.tokens) - 1):
                token.setPrev(self.tokens[i-1])
                token.setNext(Token(index=len(self.tokens)))
            else:
                token.setPrev(self.tokens[i-1])
                token.setNext(self.tokens[i+1])

    def updateAspectFields(self):
```

```
        ''' Update some token-based fields of Aspects, and aspect-based fields of Tokens
            '''
        for at in self.aspect_terms:
            # Find the aspect term within the sentence, then update the indices of the tokens.
            at.setIndices(self.findHeadIndex(at))

            # Update the tokens' IOB2 fields
            self.tokens[at.headIndex].setActualIOB2("B")
            for i in range(at.headIndex+1, at.endIndex):
                self.tokens[i].setActualIOB2("I")

            # Add a list of the Token objects for the aspect term
            at.setTokens(self.tokens[at.headIndex:at.endIndex])

    ''' NOTE: No longer needed
    def predictedFromIOB2(self):
        Given an instance with predicted IOB2 tags, return a list of predicted Aspects

        term = []
        termList = []
        i = 0
        while i < len(self.tokens):
            t = self.tokens[i]
            if t.predictedIOB2 == "B":
                term.append(t)
                while i+1 < len(self.tokens):
                    if self.tokens[i+1].predictedIOB2 == "I":
                        term.append(self.tokens[i+1])
                        i = i + 1
                    else:
                        break
                termList.append(term)
                term = []
            i = i + 1
        return termList
    '''

    def findHeadIndex(self, at):
        ''' Two challenges here: we must account for multi-word aspect terms, and we must account for duplicates of the term that may exist in the sentence. '''
        headToken = at.tokenized_term[0]          # The first token of the aspect term(if multiple tokens are in the word/phrase)
        headCount = self.text.count(headToken)    # Count how many times the first word in the aspect term appears in the sentence
```

```python
        index = -1                              # The index we're looking for - will eventually be returned

        # If there is only one occurrence of the aspect term's first word:
        if headCount == 1:
            return findNthOccurrence(self.tokenized_text, headToken, 1)

        # If there are multiple occurrences, find the correct occurrence and then find its' index in the token list
        else:
            n = 1        # The nth occurrence of the word is the one we're searching for
            loc = -1     # The current location within the sentence string
            while n <= headCount:
                # Find the next occurrence and check if it matches the listed beginning offset.
                loc = self.text.find(headToken, loc+1)
                if loc == int(at.offsets['from']):
                    # Find the index in the tokens of the nth occurrence of the term
                    return findNthOccurrence(self.tokenized_text, headToken, n)
                n = n + 1
            return -1

    def adjustFormat(self):
        ''' For evaluation purposes. Returns a list of (Term, Indices) tuples, where Indices is a tuple
            '''
        output = []
        for at in self.aspect_terms:
            term = " ".join(at.lower_tokenized_term)
            indices = tuple([token.index for token in at.tokens])
            output.append((term, indices))

        return output

    def get_aspect_terms(self):
        return [a.lower_term for a in self.aspect_terms]

    def get_aspect_categories(self):
        return [c.term.lower() for c in self.aspect_categories]

    def get_predicted_terms(self):
        return [a.lower_term for a in self.predicted_terms]
```

```python
    def get_predicted_categories(self):
        return [c.term.lower() for c in self.predicted_categories]

    def add_aspect_term(self, term, offsets='', id=''):
        a = Aspect(term, id)
        if offsets != '':
            a.setOffsets(offsets)
        self.aspect_terms.append(a)

    def add_aspect_category(self, term, polarity=''):
        c = Category(term, polarity)
        self.aspect_categories.append(c)

    def add_predicted_term(self, term, id=''):
        a = Aspect(term, id)
        self.predicted_terms.append(a)

    def add_predicted_category(self, term, polarity=''):
        c = Category(term, polarity)
        self.predicted_categories.append(c)

class Corpus:
    '''A corpus contains instances, and is useful for training algorithms or splitting to train/test files.'''

    def __init__(self, elements):
        self.corpus = [Instance(e) for e in elements]
        self.texts = [t.text for t in self.corpus]
        self.size = len(self.corpus)
        self.aspect_terms_fd = fd([" ".join(a.tokenized_term) for i in self.corpus for a in i.aspect_terms])
        self.top_aspect_terms = freq_rank(self.aspect_terms_fd)

    def __iter__(self):
        for i in self.corpus:
            yield i.tokenized_text

    def top_text_terms(self):
        ''' Old version of top_aspect_terms
            '''
        aspect_terms_fd = fd([a for i in self.corpus for a in i.get_aspect_terms()])
        return freq_rank(self.aspect_terms_fd)

    def clean_tags(self):
        for i in range(len(self.corpus)):
            self.corpus[i].aspect_terms = []

```

```
    def split(self, threshold=0.8, shuffle=False):
        '''Split to train/test, based on a threshold. Turn on shuffling for randomizing the elements beforehand.'''
        clone = copy.deepcopy(self.corpus)
        if shuffle: random.shuffle(clone)
        train = clone[:int(threshold * self.size)]
        test = clone[int(threshold * self.size):]
        return train, test

    def getPolarityTermDict(self):
        ''' Returns a dictionary where each aspect term is associated with a dictionary,
        '''
        return fd2([(at.term, at.polarity) for at in s.aspect_terms for s in self.corpus])

    def getPolarityCategoryDict(self):
        ''' Returns a dictionary where each aspect category is associated with a dictionary
        '''
        return fd2([(ac.term, ac.polarity) for ac in s.aspect_categories for s in self.corpus])
```

Appendix - Aspect Identification

```python
import time
import math
from collections import defaultdict
import xml.etree.ElementTree as ET
from libraries.structure import Corpus
from libraries.structure import fd
from libraries.structure import freq_rank
from libraries.structure import generate

from stanford_corenlp_python import jsonrpc

import nltk
import nltk.corpus, nltk.tag
from nltk import word_tokenize
from nltk.tokenize import WordPunctTokenizer
from nltk.tokenize import TreebankWordTokenizer as Tokenizer
import nltk.chunk as chunk
from nltk.stem.porter import PorterStemmer as Stemmer

import pycrfsuite

################### Association Mining Method ###################

def HuLiu(instances, minSupport = 1.0, minPsupport = 2, maxWordDist = 1.0, maxNonCompact = 1):
    ''' Hu and Liu's algorithm for aspect term extraction. Returns two arguments: a dictionary containing all predicted terms with their associated p-support, and a list of sentences with the aspect terms in each sentence.
        instances = a list of Sentence
        minSupportPercentage = the percentage of sentences the term must appear in to be considered "frequent"
        minPsupport = the minumum number of sentences in which a candidate term must occur (ignoring any times another candidate term in the sentence subsumes the current candidate term)
        maxWordDist = the maximum distance allowed between words in a candidate term
```

```
        maxNonCompact = the maximum number of sentences within the
↪   corpus in which a candidate term can violate the maximum word
↪   distance
        '''

    # We store the terms by sentence and as a set.
    terms = dict()                            # Stores terms in a
    ↪   variety of formats
    terms['sent'] = dict()                    # Stores the terms of
    ↪   each sentence separately
    terms['sent']['(Term,Indices)'] = []      # Stores the terms of
    ↪   each sentence as a tuple: (FullString, IndicesTuple). FullString
    ↪   has " " between tokens. IndicesTuple is a tuple containing the
    ↪   indices of the tokens in String.
    terms['all'] = dict()                     # Stores terms in one
    ↪   group, not as a list of sentences
    terms['all']['set'] = set()               # The entire set of
    ↪   distinct terms

    tbc = 0
    # treebank chunking
    if tbc:
        treebank_sents = nltk.corpus.treebank_chunk.chunked_sents()
        train_chunks = conll_tag_chunks(treebank_sents)
        u_chunker = nltk.tag.UnigramTagger(train_chunks)
        ub_chunker = nltk.tag.BigramTagger(train_chunks,
        ↪   backoff=u_chunker)
        ubt_chunker = nltk.tag.TrigramTagger(train_chunks,
        ↪   backoff=ub_chunker)
        ut_chunker = nltk.tag.TrigramTagger(train_chunks,
        ↪   backoff=u_chunker)
        utb_chunker = nltk.tag.BigramTagger(train_chunks,
        ↪   backoff=ut_chunker)
        # Find nouns and noun phrases in each sentence - these are
        ↪   initial candidate terms. Nouns are lists of (String, Index)
        ↪   tuples
        nounsBySentence = [nounsAndPhrasesInSentence(s,
        ↪   chunker=ub_chunker, tbc=True) for s in instances]
    else:
        # Find nouns and noun phrases in each sentence - these are
        ↪   initial candidate terms. Nouns are lists of (String, Index)
        ↪   tuples
        nounsBySentence = [nounsAndPhrasesInSentence(s, ne=True) for s
        ↪   in instances]

    # Include combined pairs and triples of nouns / phrases within
    ↪   sentences as candidate terms, then get a dictionary of their
    ↪   frequencies (support)
```

```python
    temp = [[t2list(term) for term in (s + getPairs(s) + getTriples(s))] for s in nounsBySentence]

    support = fd([term[0] for s in temp for term in s])

    # Get all frequent candidate terms - those that meet the minimum support.
    terms['sent']['(Term,Indices)'] = [[term for term in s if support[term[0]] >= minSupport] for s in temp]

    # Update support
    support = fd([term[0] for s in terms['sent']['(Term,Indices)'] for term in s])

    # Store the set of current candidate terms
    terms['all']['set'] = set(support.keys())

    nonCompact = dict.fromkeys(terms['all']['set'], 0) # Stores occurrences of non-compact form for each term
    pSupport = fd([term[0] for s in terms['sent']['(Term,Indices)'] for term in removeSubsets(s)])   # Stores p-support of each term
    isSubset = dict.fromkeys(terms['all']['set'])

    for term in terms['all']['set']:
        isSubset[term] = False
        for term2 in terms['all']['set']:
            if term in term2:
                if term != term2:
                    isSubset[term] = True
                    continue

    # Check to see if the distance between words exceeds maxDist
    for sentence in terms['sent']['(Term,Indices)']:
        for term in sentence:
            indices = term[1]
            if len(indices) <= 1:
                # Term has only one word - skip to next term
                continue
            max = maxDist(indices)
            if max > maxWordDist:
                nonCompact[term[0]] = nonCompact[term[0]] + 1

    # Remove terms that appear in non-compact form more than "maxNonCompact" times. Also, remove terms below the minimum p-support threshold that are a subset of some other term.
    newTerms = set()
    sub = 0
    nc = 0
```

```
    for term in terms['all']['set']:
        if nonCompact[term] > maxNonCompact:
            # Condition violated, term is removed
            nc += 1
            continue
        if pSupport[term] >= minPsupport:
            # Term meets minimum p-support; term is kept
            newTerms.add(term)
        else:
            if isSubset[term]:
                # Term is a part of another aspect term; term is removed
                sub += 1
                continue
            else:
                # Term is not part of another term; term is kept
                newTerms.add(term)

    terms['all']['set'] = newTerms

    # Update terms['sent']['(Term,Indices)']
    newTermSents = []
    for sentence in terms['sent']['(Term,Indices)']:
        newSent = []
        for term in sentence:
            if term[0] in terms['all']['set']:
                newSent.append(term)
        newTermSents.append(newSent)
    terms['sent']['(Term,Indices)'] = newTermSents

    # Update p-support values
    pSupport = fd([term[0] for s in terms['sent']['(Term,Indices)'] for
    ↪ term in removeSubsets(s)])   # Stores p-support of each term

    return support, terms['sent']['(Term,Indices)']

def nearestNoun(sentence, adjIndex):
    ''' Returns the nearest noun to a given term in a sentence. Sentence
    ↪ is an Instance, adjIndex is the index of the adjective in the
    ↪ sentence.
        Returns None if there are no nouns in the sentence'''
    nouns = [(token.text, token.index) for token in sentence if
    ↪ token.POS2 == "NN"]
    if len(nouns) == 0:
        return None
    adjIndex = adj[1]
    nearest = None
    minDist = float("inf")
    for noun in nouns:
```

```
        dist = abs(adjIndex - noun[1])
        if dist < minDist:
            minDist = dist
            nearest = noun
    return nearest

def maxDist(indices):
    ''' Takes a list of indices (ex: [1, 3, 4, 5]
        Returns the max distance between any 2 adjacent indices '''
    maxDist = 0
    for i in range(len(indices)-1):
        dist = indices[i+1] - indices[i]
        if dist > maxDist:
            maxDist = dist
    return maxDist

def neChunker(instance):
    tagged = [(token.text, token.POS) for token in instance.tokens]
    rawChunks = nltk.chunk.ne_chunk(tagged)
    (tags, chunks) = zip(*(conll_tag_chunks([rawChunks])[0]))
    return chunks

def nounsAndPhrasesInSentence(instance, chunker='', reg=False, tbc=False, ne=False):
    ''' Input: A sentence instance
        Returns a list of lists, each containing the (String, Index) tuples corresponding to the tokens of a noun or noun phrase
        '''
    # Get tagged sentence in the form of a list of (token, POS) tuples
    tagged = [(token.text, token.POS) for token in instance.tokens]
    # Create a list of (String, Index) tuples for each noun / noun phrase
    nouns = []
    if reg==True:
        # Find noun phrases using regex
        pattern = r"""
            NBAR:
            {<NN.*|JJ>*<NN.*>}  # Nouns and Adjectives, terminated with Nouns
            NP:
            {<NBAR>}
            {<NBAR><IN><NBAR>}  # Above, connected with in/of/etc...
            """
        NPChunker = nltk.RegexpParser(pattern)
        tagged = NPChunker.parse(tagged)
        # cTagged = chunk.ne_chunk(tagged)
        nounIndices = chunkParse(tagged)

```

```
        for n in nounIndices:
            nList = []
            for i in n:
                nList.append((instance.tokens[i].lower_text, instance.tokens[i].index))
            nouns.append(nList)
    elif tbc==True and chunker == '':
        print "ERROR: Chunker must be provided."
        exit()
    else:
        if ne == True:
            chunks = neChunker(instance)
        elif tbc == True:
            (words, tags) = zip(*tagged)
            (tags2, chunks) = zip(*chunker.tag(tags))
        else:
            print "Error"
            exit()
        n = []

        # Iterate over tokens
        for i in range(len(instance.tokens)):
            # Add nouns outside of noun phrases
            if chunks[i] == 'O':
                if tagged[i][1].startswith('NN'):
                    nouns.append([(instance.tokens[i].lower_text, i)])
            # Start or continue building noun phrase
            else:
                n.append((instance.tokens[i].lower_text, i))

                # Check if current token is the last token
                if i+1 >= len(instance.tokens):
                    # If we were building a noun, add it
                    if len(n) > 0:
                        nouns.append(n)
                        n = []
                #
                elif chunks[i+1].startswith('I') == False:
                    nouns.append(n)
                    n = []

    return nouns

def getPairs(terms):
    ''' Given a list of terms stored as lists of (String, Index) tuples, return all pairs (e.g. [1,2], [2,3], [3,4], etc.)
        '''
    pairs = []
```

```python
    if len(terms) >= 2:
        for i in range(len(terms)-1):
            pairs.append(terms[i] + terms[i+1])
    return pairs

def getTriples(terms):
    ''' Given a list of terms stored as lists of (String, Index) tuples, return all triples (e.g. [1,2,3], [2,3,4], etc.)
        Input example:  [ [('Microsoft',2), ('Office',3)], [('Word',5)],  [('Key',8), ('Board',9)] ]
        Output example: [ [('Microsoft',2), ('Office',3), ('Word',5)], [('Word',5), ('Key',8), ('Board',9)] ]
        '''
    triples = []
    if len(terms) >= 3:
        for i in range(len(terms)-2):
            triples.append(terms[i] + terms[i+1] + terms[i+2])
    return triples

def removeSubsets(terms):
    ''' Given a list of (Term, Indices) tuples, return a list of (Term, Indices) tuples without any subsets (strings that are substrings of another string in the list)
        '''
    newTerms = []
    for i in range(len(terms)):
        subset = False
        for j in range(len(terms)):
            if i != j and set(terms[i][1]) < set(terms[j][1]):
                    subset = True
        if subset == False:
            newTerms.append(terms[i])
    return newTerms

def t2list(term):
    ''' Given a term stored as a list of (String, Index) tuples, return a tuple: (Full_String, Index_Tuple)
        '''
    if len(term) == 0:
        return term
    t = zip(*term)
    return (" ".join(t[0]), t[1])

def chunkParse(cTagged):
```

```
    ''' Given a sentence tagged with chunks (using nltk.pos_tag and RegexpParser), return a list. Each element is a list itself, containing the indices of each noun / noun phrase; single-element lists are a single index and correspond to single-word nouns. Multi-element lists store a list of indices in sequence, and are noun phrases.
        '''
    index = 0
    parsed = []
    for i in range(len(cTagged)):
        if type(cTagged[i]) is nltk.tree.Tree:
            parsed.append(range(index, index + len(cTagged[i])))
            index = index + len(cTagged[i])
        else:
            if cTagged[i][0].startswith('NN'):
                parsed.append([index])
            index = index + 1
    return parsed

def conll_tag_chunks(chunk_sents):
    tag_sents = [nltk.chunk.tree2conlltags(tree) for tree in chunk_sents]
    return [[(t, c) for (w, t, c) in chunk_tags] for chunk_tags in tag_sents]

#################### Conditional Random Fields ####################

def crf(train, test, alg = ""):
    # Convert sentences to appropriate feature/label format
    train_features, train_labels = generate(train)
    test_features, test_labels = generate(test)

    # Train CRF
    trainer = pycrfsuite.Trainer()
    for x,y in zip(train_features, train_labels):
        trainer.append(x,y)

    if alg != "":
        trainer.select(alg)
    trainer.set_params({
                       # 'c1': 1.0,   # coefficient for L1 penalty
                       # 'c2': 1e-3,  # coefficient for L2 penalty
                       # 'max_iterations': 50,  # stop earlier

                       # include transitions that are possible, but not observed
                       'feature.possible_transitions': False
                       })
```

```
    trainer.train('conll2002-esp.crfsuite')
    trainer.logparser.last_iteration

    # Test CRF
    tagger = pycrfsuite.Tagger()
    tagger.open('conll2002-esp.crfsuite')

    predicted_labels = [tagger.tag(s) for s in test_features]
    print "# Sentences: " + str(len(test_labels))
    confusion = dict()
    confusion['B'] = {'actual':0, 'predicted':0, 'truePos':0, 'falseNeg':0, 'falsePos':0}
    confusion['I'] = {'actual':0, 'predicted':0, 'truePos':0, 'falseNeg':0, 'falsePos':0}
    confusion['O'] = {'actual':0, 'predicted':0, 'truePos':0, 'falseNeg':0, 'falsePos':0}
    for s in range(len(test_labels)):
        for t in range(len(test_labels[s])):
            actual = test_labels[s][t]
            pred = predicted_labels[s][t]
            confusion[actual]['actual'] = confusion[actual]['actual'] + 1
            confusion[pred]['predicted'] = confusion[pred]['predicted'] + 1
            if actual == pred:
                confusion[actual]['truePos'] = confusion[actual]['truePos'] + 1
            else:
                confusion[actual]['falseNeg'] = confusion[actual]['falseNeg'] + 1
                confusion[pred]['falsePos'] = confusion[pred]['falsePos'] + 1
    return predicted_labels


#################### ATE Evaluation Methods ####################

# Evaluate ATE methods
def evaluateAspectTerms(trueFD, trueSent, predictedFDs, predictedSents, methodNames = [], toScreen = False):
    ''' Evaluate ATE methods
        '''

    distinct = {"TP":{}, "FN":{}, "FP":{}, "P":{}, "R":{}, "F":{}}
    instances = {"TP":{}, "FN":{}, "FP":{}, "P":{}, "R":{}, "F":{}}
    weighted = {}
    if len(methodNames) == 0:
        methodNames = [str(i) for i in range(1,len(predictedFDs)+1)]
```

```
    for i in range(len(predictedFDs)):
        predictedFD = predictedFDs[i]
        predictedSent = predictedSents[i]

        # Distinct
        TP, FN, FP, P, R, F = evaluateAspectTermsDistinct(set(trueFD.keys()), set(predictedFD.keys()))
        distinct["TP"][methodNames[i]] = TP
        distinct["FN"][methodNames[i]] = FN
        distinct["FP"][methodNames[i]] = FP
        distinct["P"][methodNames[i]] = P
        distinct["R"][methodNames[i]] = R
        distinct["F"][methodNames[i]] = F

        # Instances
        TP, FN, FP, P, R, F = evaluateAspectTermsInstances(trueSent, predictedSent, trueFD, predictedFD)
        instances["TP"][methodNames[i]] = TP
        instances["FN"][methodNames[i]] = FN
        instances["FP"][methodNames[i]] = FP
        instances["P"][methodNames[i]] = P
        instances["R"][methodNames[i]] = R
        instances["F"][methodNames[i]] = F

        # Average weighted precision
        weighted[methodNames[i]] = evaluateAspectTermsWeighted(trueFD, predictedFD)

    if toScreen:
        print "------------------- Distinct: --------------------"
        for m in methodNames:
            print m
            print "TP: " + str(distinct["TP"][m])
            print "FN: " + str(distinct["FN"][m])
            print "FP: " + str(distinct["FP"][m])
            print "P: " + str(distinct["P"][m])
            print "R: " + str(distinct["R"][m])
            print "F: " + str(distinct["F"][m])
            print ""

        print "------------------- Instances: --------------------"
        for m in methodNames:
            print m
            print "TP: " + str(instances["TP"][m])
            print "FN: " + str(instances["FN"][m])
            print "FP: " + str(instances["FP"][m])
            print "P: " + str(instances["P"][m])
```

```
            print "R: " + str(instances["R"][m])
            print "F: " + str(instances["F"][m])
            print ""

        print "------------------- Average Weighted Precision: -------------------"
        for m in methodNames:
            print m
            print str(weighted[m])
            print ""

    return distinct, instances, weighted

# Evaluate by distinct aspect terms
def evaluateAspectTermsDistinct(trueSet, predictedSet, toScreen = False):
    ''' Input: two sets of distinct aspect terms (the predicted set and the true set) for a corpus
        Output: Evaluation metrics for distinct aspect terms'''

    truePos = predictedSet.intersection(trueSet)
    falseNeg = trueSet.difference(predictedSet)
    falsePos = predictedSet.difference(trueSet)
    TP = len(truePos)
    FN = len(falseNeg)
    FP = len(falsePos)
    if TP == 0:
        P = 0.0
        R = 0.0
        F = 0.0
    else:
        P = float(TP)/(TP + FP)
        R = float(TP)/(TP + FN)
        F = 2.0*P*R/(P+R)

    '''
    print "\nEvaluate by Distinct Term: "
    print "Predicted: " + str(len(ptSet))
    print "Actual: " + str(len(atSet))
    '''

    if toScreen:
        print "True Positive: %f -- False Negative: %f -- False Positive: %f (P = %d, R = %d, F = %d)" % TP, FN, FP, P, R, F

    return TP, FN, FP, P, R, F
```

```
def evaluateAspectTermsInstances(trueTermsBySent, predictedTermsBySent, trueFD="", predictedFD="", toScreen = False):
    ''' Input:  predictedTerms (a list of sentences, where each sentence is expressed as a list of its predicted aspect terms in (Term, Indices) format) and actualTerms (same as predictedTerms, but with the human-annotated terms). If true and predicted frequency dictionaries are not specified, they are computed.
        Output: Evaluation metrics for instances of aspect terms'''

    if trueFD == "":
        trueFD = fd([term[0] for s in trueTermsBySent for term in s])
    if predictedFD == "":
        predictedFD = fd([term[0] for s in predictedTermsBySent for term in s])

    # Below is a version of the code that gives frequencies for true positive, false negative, and false positive for each word:

    truePosList = []    # List for terms in both actual and predicted
    falseNegList = []   # List for terms in actual but not predicted
    falsePosList = []   # List for terms in predicted but not actual

    # Check whether terms are in actual, predicted, or both
    for i in range(len(trueTermsBySent)):
        # Get true and predicted sets of term indices tuples from sentence
        trueIndices = set([term[1] for term in trueTermsBySent[i]])
        predictedIndices = set([term[1] for term in predictedTermsBySent[i]])

        # Update TP, FN, and FP using sets
        truePosList.extend([term[0] for term in trueTermsBySent[i] if term[1] in trueIndices.intersection(predictedIndices)])
        falseNegList.extend([term[0] for term in trueTermsBySent[i] if term[1] in trueIndices.difference(predictedIndices)])
        falsePosList.extend([term[0] for term in predictedTermsBySent[i] if term[1] in predictedIndices.difference(trueIndices)])

    truePos = fd(truePosList)
    falseNeg = fd(falseNegList)
    falsePos = fd(falsePosList)
    '''
    print "------TruePos-------"
    print list(truePos.keys())[:100]
    print "------FalseNeg-------"
    print list(falseNeg.keys())[:100]
    print "------FalsePos-------"
    print list(falsePos.keys())[:100]
```

```
    '''

    TP = sum(truePos.values())
    FN = sum(falseNeg.values())
    FP = sum(falsePos.values())

    if TP == 0:
        P = 0.0
        R = 0.0
        F = 0.0
    else:
        P = float(TP)/(TP + FP)
        R = float(TP)/(TP + FN)
        F = 2.0*P*R/(P+R)

    '''
    print "\nEvaluate by Instance: "
    print "Predicted: " + str(sum(train_data.predicted_terms_fd.values()))
    print "Actual: " + str(sum(train_data.aspect_terms_fd.values()))
    '''
    if toScreen:
        print "True Positive: %f -- False Negative: %f -- False Positive: %f (P = %d, R = %d, F = %d)" % TP, FN, FP, P, R, F

    return TP, FN, FP, P, R, F

def evaluateAspectTermsWeighted(trueFD, predictedFD, toScreen = False):
    ''' Inputs: dictionaries of term frequencies, both true and predicted. toScreen specifies whether to print output or not
        Outputs: the average weighted precision
        '''
    trueFD_sorted = freq_rank(trueFD)
    trueRanked = {trueFD_sorted[i]:(i+1) for i in range(len(trueFD_sorted))}
    awp = avgWeightedPrecision(trueRanked, freq_rank(predictedFD))
    if toScreen:
        print "Average weighted precision: " + str(awp)
    return awp

def weightedPrecision(trueSet, predictedFreqRank, m):
    ''' Inputs: a set of true aspect terms, a list of predicted terms (in order of decreasing frequency), and a parameter m.
        Output: the weighted precision of the first m predicted terms.
        '''
    predicted = predictedFreqRank[0:m]
    wp = 0.0
    denom = 0.0
```

```
    for i in range(m):
        denom = denom + 1.0/(i+1.0)
        if predicted[i] in trueSet:
            wp = wp + 1.0/(i+1.0)
    return wp/denom

def weightedRecall(trueRanked, predictedFreqRank, m):
    ''' Inputs: a dictionary of true aspect terms, where values are their frequency rank (ex: the kth most frequent term has value k), a list of predicted terms, and a parameter m.
        Output: the weighted recall of the first m predicted terms.
        '''
    predicted = predictedFreqRank[0:m]
    wr = 0.0
    denom = 0.0
    # Compute the numerator
    for i in range(m):
        if predicted[i] in trueRanked:
            # Sum the reciprocal of the
            wr = wr + 1.0/trueRanked[predicted[i]]
    # Compute the denominator
    for i in range(len(trueRanked)):
        denom = denom + 1.0/(i+1.0)
    return wr/denom

def avgWeightedPrecision(trueRanked, predictedFreqRank):
    ''' Inputs: a dictionary of true aspect terms, where values are their frequency rank (ex: the kth most frequent term has value k), a list of predicted terms, and a parameter m.
        Output: the weighted recall of the first m predicted terms.
        '''
    awp = 0.0
    wr = [weightedRecall(trueRanked, predictedFreqRank, m) for m in range(1, len(predictedFreqRank) + 1)]
    wp = [weightedPrecision(trueRanked, predictedFreqRank, m) for m in range(1, len(predictedFreqRank) + 1)]
    for i in range(11):
        r = i/10.0
        max = 0.0
        for m in range(0, len(predictedFreqRank)):
            if wr[m] >= r:
                if wp[m] > max:
                    max = wp[m]
        awp = awp + max
    return awp

#################### Data Processing ####################
```

```python
def IOB2toAspectTerms(IOB2labels, sentences):
    ''' Input: A list of IOB2 labels corresponding to sentences, and a list of sentences (Instance objects)
        Output: support (a dictionary of aspect terms, each token separated with " ", and frequencies) and predictedSentences (a list of sentences, each stored as a list of (Term, Indices) tuples.
        '''
    predictedTermsBySentence = []
    for i in range(len(sentences)):
        predictedTermsBySentence.append([])
        labels = IOB2labels[i]
        sentence = sentences[i]
        term = ""
        indices = []
        for j in range(len(sentence.tokens)):
            token = sentence.tokens[j]
            if labels[j] == "B":
                term = token.lower_text
                indices.append(token.index)
            if labels[j] == "I":
                term = term + " " + token.lower_text
                indices.append(token.index)
            if ((j+1) == len(sentence.tokens)) and (len(term) > 0):
                predictedTermsBySentence[i].append((term, tuple(indices)))
                term = ""
                indices = []
                continue
            if (labels[j] != "O") and (labels[j+1] != "I"):
                predictedTermsBySentence[i].append((term, tuple(indices)))
                term = ""
                indices = []
    support = fd([term[0] for s in predictedTermsBySentence for term in s])
    return support, predictedTermsBySentence
```

Appendix - Sentiment Analysis

```python
import time
from collections import defaultdict
import xml.etree.ElementTree as ET
from libraries.structure import Corpus
from libraries.structure import fd
from libraries.structure import freq_rank
from libraries.structure import generate

from stanford_corenlp_python import jsonrpc

import nltk
from nltk import word_tokenize
from nltk.tokenize import WordPunctTokenizer
from nltk.tokenize import TreebankWordTokenizer as Tokenizer
from nltk.corpus import sentiwordnet as swn
import nltk.chunk as chunk
from nltk.stem.porter import PorterStemmer as Stemmer
from nltk.corpus import wordnet as wn
from nltk.sentiment.vader import SentimentIntensityAnalyzer

import pycrfsuite

#################### Extracting adjectives from text ####################

def getOpinionAdjs(instances, terms):
    ''' Input: a list of Instances (corpus) and terms (organized as a list of sentences, where each sentence is a list of terms with the format (Term, Indices)
        Output: the nearest opinion adjective to each term, organized as a dictionary where frequency is the value
        '''
    # Create a dictionary of opinion adjectives
    opinionAdjs = defaultdict(lambda:0)
    for i in range(len(terms)):
        sentence = terms[i]
        for term in sentence:
```

```
            adj = nearestAdj(instances[i], term)
            if adj != None:
                opinionAdjs[adj[0]] = opinionAdjs[adj[0]] + 1
    return opinionAdjs

def getTermsFromAdjs(sentences, terms, opinionAdjs):
    ''' INCOMPLETE
        '''
    # Update candidate terms based on set of adjectives
    for i in range(len(terms['sent']['(Term,Indices)'])):
        # Check if there are any candidate terms - if so, move on
        if len(terms['sent']['(Term,Indices)'][i]) > 0:
            continue
        sentence = sentences['ins'][i]
        adjs = [(token.text, token.index) for token in sentence.tokens if (token.POS2 == "JJ" and token.text in opinionAdjs)]
        for a in adjs:
            noun = nearestNoun(sentence, a)
            # If there are multiple opinion adjectives in a sentence, they may return the same nearest noun
            if noun != None and noun not in terms['sent']['(Term,Indices)'][i]:
                terms['all']['set'].add(noun)
                terms['sent']['(Term,Indices)'][i].append(noun)
    return terms

def nearestAdj(sentence, term):
    ''' Returns the nearest adjective as a tuple (Adj, Index) to a given term in a sentence. Sentence is a list of Token objects, term is a (Term, Indices) tuple.
        Returns "" if there are no viable adjectives in the sentence'''
    # Find all adjectives in sentence
    adjs = [(token.text, token.index) for token in sentence if token.POS2 == "JJ"]
    if len(adjs) == 0:
        return None
    # Find the term's "average" index value (ex: a term with indices [1, 2, 3, 5] would have a center of 3.75)
    termIndices = map(list, term[1])
    termAvgIndex = sum(termIndices) / float(len(termIndices))
    # Find the adjective closest to the term's "average" index
    nearest = ""
    minDist = float("inf")
    for adj in adjs:
        # Don't count an adjective if it's already within the candidate term
        if adj in term:
            continue
```

```
        dist = abs(termAvgIndex - adj[1])
        if dist < minDist:
            minDist = dist
            nearest = adj
    return nearest

#################### Extracting polarity scores ####################

def polaritiesByCluster(polarityDict, clusters):
    clusterPolarityDict = {}
    for c in clusters:
        clusterPolarityDict[c] = defaultdict(lambda: 0)
        for term in clusters[c]:
            for polType in polarityDict[term]:
                clusterPolarityDict[c][polType] += polarityDict[term][polType]
    return clusterPolarityDict

def getTermPolarities(instances):
    ''' Input: A list of instances
        Output: A dictionary of terms, where each term contains a dictionary with counts for each polarity category (positive, negative, neutral, conflict)
        '''
    return [[(aspect.term, aspect.polarity) for aspect in instance.aspect_terms] for instance in instances]

def getCategoryPolarities(instances):
    ''' Input: A list of instances
        Output: A list of sentences, where each sentence is a list of (category, polarity) tuples, where polarity is one of (positive, negative, neutral, conflict)
        '''
    return [[(category.term, category.polarity) for category in instance.aspect_categories] for instance in instances]

#################### VADER ####################

def vader(sia, sentence):
    polarity = sia.polarity_scores(sentence)
    return polarity

def vaderAdjusted(sia, sentence):
    polarity = vader(sia, sentence)
    if polarity['compound'] < 0:
        return 'negative'
    elif polarity['neu'] == 1.0:
        return 'neutral'
```

```python
        #elif abs(polarity['pos'] - polarity['neg']) < 0.05:
        #return 'conflict'
    elif polarity['pos'] > polarity['neg']:
        return 'positive'
    else:
        return 'negative'

def vaderAdjusted2(sia, instance, aspect):
    polarity = vader(sia, context(instance, aspect))
    if polarity['compound'] < 0:
        return 'negative'
    elif polarity['neu'] == 1.0:
        return 'neutral'
    #elif abs(polarity['pos'] - polarity['neg']) < 0.05:
    #return 'conflict'
    elif polarity['pos'] > polarity['neg']:
        return 'positive'
    else:
        return 'negative'

def context(instance, aspect, r=12):
    avgIndex = aspect.headIndex + (aspect.termSize - 1)/2
    beg = max(avgIndex - r, 0)
    end = min(avgIndex + r, len(instance.tokens))
    return " ".join([token.text for token in instance.tokens[beg:end]])

def vaderTermPolarities(instances, adjusted = True):
    ''' Input: A list of instances
        Output: A list of sentences, where each sentence contains a list of (term, polarity) tuples. These polarities are estimated from the VADER sentiment analyzer.
        '''
    polarities = []
    sia = SentimentIntensityAnalyzer()
    for instance in instances:
        '''
        if adjusted:
            p = vaderAdjusted(sia, instance.text)
        else:
            p = vader(sia, instance.text)
        polarities.append([(aspect.term, p) for aspect in instance.aspect_terms])
        '''
        polarities.append([(aspect.term, vaderAdjusted2(sia, instance, aspect)) for aspect in instance.aspect_terms])
    return polarities

def vaderCategoryPolarities(instances, adjusted = True):
```

```python
    ''' Input: A list of instances
        Output: A list of sentences, where each sentence contains a list of (category, polarity) tuples. These polarities are estimated from the VADER sentiment analyzer.
        '''
    polarities = []
    sia = SentimentIntensityAnalyzer()
    for instance in instances:
        if adjusted:
            p = vaderAdjusted(sia, instance.text)
        else:
            p = vader(sia, instance.text)
        polarities.append([(category.term, p) for category in instance.aspect_categories])
    return polarities

################### Ratings ###################

def computeRatingsVader(polarityDict):
    ''' Inputs: a dictionary of aspect terms/categories or clusters, where each value is a dictionary describing aggregate polarity scores (ex: {"keyboard":{"positive":4.534, "negative":2.386, "
        '''
    ratings = {}
    for term in polarityDict:
        p = polarityDict[term]
        ratings[term] = 4.0*p["positive"] / (p["positive"] + p["negative"]) + 1
    return ratings

def computeRatings(polarityDict):
    ''' Inputs: a dictionary of aspect terms/categories or clusters, where each value is a dictionary describing polarity counts (ex: {"keyboard":{"positive":5,"negative":7,"neutral":2,"conflict":1}})

        Outputs: Ratings are scored as follows: 4 * ( (P + 0.5*C)/(P + N + 0.5*C) ) + 1
        '''
    ratings = {}
    for t in polarityDict:
        p = polarityDict[t]
        ratings[t] = 4.0*(float(p["positive"] + 0.5*p["conflict"]) / (p["positive"] + p["negative"] + p["conflict"])) + 1
    return ratings

################### ABSA Evaluation Methods ###################

def evaluatePolarities(trueBySent, predictedBySent):
```

```
    # Create dictionary where confusion[i][j] is the count where a term/category with true polarity i is predicted to have polarity j.
    confusion = defaultdict(lambda:defaultdict(lambda:0))
    tot = 0
    for i in range(len(trueBySent)):
        for j in range(len(trueBySent[i])):
            confusion[trueBySent[i][j][1]][predictedBySent[i][j][1]] += 1
            tot += 1
    polTypes = ['positive', 'negative', 'neutral'] #, 'conflict']
    tot -= sum([confusion['conflict'][j] for j in polTypes])
    print confusion
    accuracy = sum(confusion[i][i] for i in polTypes) / float(tot)
    print accuracy
    precision = {i:float(confusion[i][i])/sum([confusion[i][j] for j in polTypes]) for i in polTypes}
    recall = {i:float(confusion[i][i])/sum([confusion[j][i] for j in polTypes]) for i in polTypes}
    f = {i:(2.0*precision[i]*recall[i]/(precision[i]+recall[i])) for i in polTypes}
    print "Precision:"
    print precision
    print "Recall:"
    print recall
    print "F-measure:"
    print f

def evaluateRatings(trueRatings, predictedRatings):
    ''' Input: true and predicted ratings in a dictionary (keys are terms or cluster labels, values are ratings)
        Output: Evaluation metrics
        '''
    if trueRatings.keys() != predictedRatings.keys():
        print "Error: keys don't match"

    diffs = [abs(trueRatings[t] - predictedRatings[t]) for t in trueRatings]
    MSE = sum([d^2 for d in diffs])/float(len(trueRatings))
    print "Number of terms/clusters: %d", len(trueRatings)
    print "MSE: %f", MSE
```

Biography

Sean Byrne was born in 1994 in the state of Pennsylvania. He attended Lehigh University for his undergraduate education, and was highly involved in the Industrial & Systems Engineering department through various projects with professors and as a member of the ISE Student Council. He graduated from Lehigh University with a Bachelor of Science in Industrial & Systems Engineering and a Bachelor of Science in Mathematics in May 2016. He is now completing a Master of Science degree in Industrial & Systems Engineering through the President's Scholars program, and will graduate in May 2017.