Spring 2017

NAMED ENTITY RECOGNITION AND CLASSIFICATION FOR NATURAL LANGUAGE INPUTS AT SCALE

Shreeraj Dabholkar
San Jose State University

NAMED ENTITY RECOGNITION AND CLASSIFICATION FOR NATURAL LANGUAGE INPUTS AT SCALE

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfilment

Of the Requirements for the Degree

Master of Science in Computer Science

By

Shreeraj Dabholkar

May 2017

The designated Project Committee Approves the Project Titled

Named Entity Recognition and Classification for Natural Language Inputs at Scale

By

Shreeraj Dabholkar

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2017

Dr. Thanh D. Tran (Department of Computer Science)

Dr. Thomas Austin (Department of Computer Science)

Mr. Hemang Nadkarni (Principal Engineer, McAfee LLC.)

ABSTRACT

Natural language processing (NLP) is a technique by which computers can analyze, understand, and derive meaning from human language. Phrases in a body of natural text that represent names, such as those of persons, organizations or locations are referred to as named entities. Identifying and categorizing these named entities is still a challenging task, research on which, has been carried out for many years. In this project, we build a supervised learning based classifier which can perform named entity recognition and classification (NERC) on input text and implement it as part of a chatbot application. The implementation is then scaled out to handle very high-velocity concurrent inputs and deployed on two clusters of different sizes. We evaluate performance for various input loads and configurations and compare observations to determine an optimal environment.

TABLE OF CONTENTS

I. INTRODUCTION

Languages such as English used by humans for communication are widely different from artificial languages like programming languages that are used to give instructions to machines. Natural language processing (NLP) is the process modelling the characteristics of natural language using statistical techniques [1]. It provides a way for computers to analyze, understand, and derive meaning from human language. Although it may be relatively easy for a human to recognize and classify names of places or people in each text, it is not straightforward for a machine to recognize such phrases [2]. Named entity recognition and classification (NERC) is the task of processing text to identify and classify names, enabling the extraction of useful information from documents.

A lot of modern use cases for NLP and NERC, deal with high-volume, high-velocity data. A lot of this data, like that coming from social media, is also unstructured. Single machine systems cannot handle data at this scale. A way to handle this problem is to implement a distributed multi-node solution. Such a system needs to scale to varying loads depending on the use case. Thus, it is challenging to build a named entity classifier with a high accuracy and which can scale to high volume, high velocity, simultaneous inputs.

In this project, we design a machine learning based classifier that can disambiguate named entities from natural language text and implement it as part of an NLP pipeline. Classifier performance is evaluated using metrics such as accuracy, precision, recall. We then focus on pipeline scalability for high-velocity inputs. The number of simultaneous requests a web application can serve is limited by its hardware resources. We use a distributed computing framework to process high-velocity, simultaneous inputs. We test our solution at different loads and configurations and observe performance for each.

In this paper, we start by covering an overview of the landscape of NER research conducted in recent times and the different categories of approaches to NER. We then put forth the problem definition and our proposed solution for the problems. The architecture overview covers implementation design and information about the various components of our technology stack. The implementation details section covers the process of building the classifier, the middleware module, and the distributed application in depth. We then describe load testing experiments performed on the distributed implementation and discuss results and conclusions based on our observations. The last section of this paper discusses future directions to improve the application and how the application could be used as a component in a larger, more complex data pipeline.

II. BACKGROUND & LITERATURE REVIEW

Early systems made use of hand-crafted rules and pattern matching based on these annotations and formation patterns. These are referred to as knowledge-based systems. Developing hand-crafted rules is laborious and time-consuming. The domain plays an important part in creating rules. As demonstrated in [3], The rules that are created for a domain are difficult to generalize to other domains. The other paradigm for NERC systems is based on machine learning. Recent learning based systems use models which can be classified under supervised, semi-supervised or unsupervised learning.

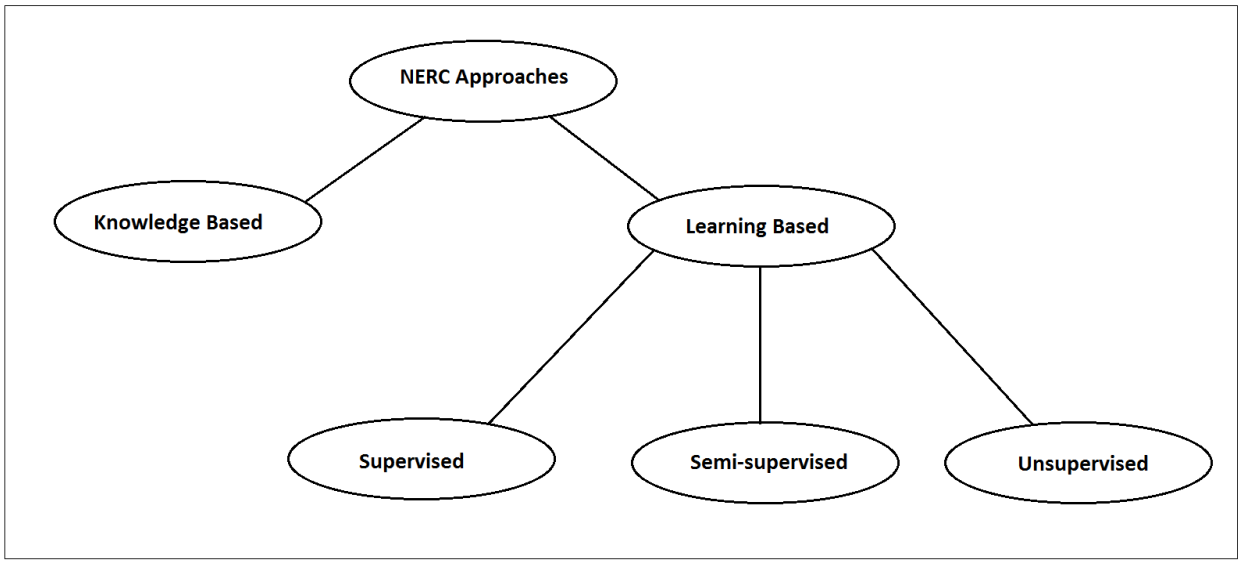

Fig. 1. Categorical Overview of NERC Approaches.

A. Named Entity Types

The definition of “Named Entities” was specified for the Sixth Message Understanding Conference (MUC-6) [6]. In order to extracting structured information from unstructured natural language sources, it is vital to recognize elements of data such as words signifying names of people, organizations, places, and, numbers like dates and monetary denominations [5]. Initial research was conducted on identifying proper nouns in a body of text as it constituted distinction of named entities from non-named entities. Systems such as the FUNES [7] recognize proper names from natural language sources in English. The most commonly used categories for classification of named entities are “Person,” “Location” and “Organization” as defined in the CoNLL-2003 shared task [4]. These three classes are collectively referred to as “ENAMEX” [6]. The “Miscellaneous” type is used to classify named entities, not under “ENAMEX.” These categories apply to multiple domains. However, a more domain specific hierarchy can yield better results [8] than a generalized one.

B. Knowledge based systems

Knowledge-based systems recognize and classify named entities from text based by making use of dictionaries known as gazetteers along with handcrafted heuristics using pattern matching techniques. As they function on a set of predefined rules, the classifications are

transparent and can be expanded when new, previously unseen inputs are encountered. With these systems, there is a better visibility into the internal state of the system which results in easier error analysis. The process of creating gazetteers and building rule sets involves considerable manual effort like in [9] where multiple dictionaries, including one containing common nouns with about 50,000 instances was created to classify named entities encountered into 200 fine-grained categories. Entities that cannot be tagged by the dictionary are tagged based on specific rules [9].

There are two pitfalls associated with dictionary based NERC systems. First, there are usually many misclassifications due to short names which cause the precision score to fall. Second, spelling variations can cause the recall score to be relatively low. These potential pitfalls can effectively lower the F-score, a combination of precision and recall, resulting in a system that evaluates poorly. Another challenging problem with rule-based NERC is domain customization. Generalizing such a system designed for one domain over for other domains requires a significant manual effort. Authors of [10] define “a high-level rule language” for developing classifiers used for NERC and leverage it to build a complex rule based annotator CoreNER, which can be customized to different domains.

C. Learning based systems

Many modern NERC systems leverage Machine Learning (ML). These involve providing sets of positive and negative training sets to a statistical model, which then "learns" patterns from the input. A portion of the data is usually held back from training and used to evaluate the trained model. Modern ML based classifiers fall under one of three discrete types, supervised learning, semi-supervised learning, and unsupervised learning.

1) Supervised Learning:

Supervised learning methods make use of training datasets which include classification labels for each data point. A model is trained by feeding it numerous positive and negative classification instances. It then establishes learned rules and leverages them to make predictions on new data. These techniques require the use of a large annotated corpus as a source of training and testing data. Creation of such corpora requires extensive manual effort, and as such, they are not easily available for free. This often creates the need for the manual creation of an annotated corpora, an intricate and tedious task. Various supervised learning algorithms have been implemented for NERC such has Hidden Markov Models, Support Vector Machines, Conditional Random Fields, Maximum Entropy performing the best amongst others [5]. Domain customization for a supervised learning system involves acquiring labeled data for the new domain and learning a new model from scratch.

2) Semi-supervised Learning:

Semi-supervised learning techniques are relatively recent and are essentially a hybrid of supervised and unsupervised approaches. They usually involve a small set of example names referred to as “seeds” being provided to the system. Sentences containing these names are then searched, and the system identifies contextual clues that may be common to these names. The system then searches for other names that appear in similar contexts. Once a few such names are found, the learning process is reapplied to the new extended set of names and with each iteration, more names are identified. Riloff and Jones [11] introduce the concept of “mutual bootstrapping,” a technique that uses the seed words to learn extraction patterns and then exploits the learned extraction patterns to identify more words that belong to that semantic category [20].

3) Unsupervised Learning:

Unsupervised learning techniques deal with the NERC problem by resolving it into a clustering one. They do not require initial training data to be fed to them which makes them a popular approach for resource-starved languages and domains [12]. Clustering is carried out by aggregating named entities into contextual groups. There are no labels provided at the beginning of an unsupervised algorithm. When the algorithm finishes, it

outputs groups of entities that share similar features. One approach to labeling obtained clusters of named entities is to attribute a label to a subject for each group [13]. This property is useful in cases such as the simultaneous appearance of named entities in multiple news articles [14].

III. PROBLEM DEFINITION

This project has two primary goals. The first goal is to design a supervised learning based classifier for named entity disambiguation. Classifier performance should be evaluated and optimized. We also want to implement the classifier in an example application to demonstrate a real-world use case and load test the classifier's ability to handle a high number of incoming requests.

The second goal is to implement the core named entity classifier as a part of a scalable, distributed computing application which can handle simultaneous, high-velocity inputs. The performance of the application should be observed at varying input rates and different configurations so that inferences may be drawn by comparing them.

IV. PROPOSED SOLUTION

A Supervised learning based NER system will be implemented using the Maximum Entropy model. The domain for training and test data will be established. Once the model is trained on the training set, it will be used to make predictions on the testing set and performance will be assessed against the baseline using accuracy, precision, recall and f-measure as metrics. Then, the performance of the model with respect to scoring metrics, as well as time taken for training and predicting, will be optimized. The classifier will be wrapped into a REST based API and plugged in as a middleware module into an example Chat Bot application to demonstrate a real-world use case. The middleware module will process conversational text between a user and the Chat Bot, internally using our classifier.

We will implement a multi-node, in-memory cluster computing framework which will distribute both data and compute over multiple nodes to handle high-velocity, concurrent requests. Incoming requests will be channeled into a data stream and plugged into our distributed classifier application as a continuous input. The application will output named entity classifications for each request. This solution will be deployed on a cloud-based infrastructure to ensure and manage scalability. We will evaluate effects of various loads and optimizations on performance.

V. IMPLEMENTATION DETAILS

A. The MaxEnt Classifier

We use two datasets to serve as training and test sets. 75% of each set is used to train the models, and 25% is used to verify predictions.

Wiki Gold dataset: Contains around 35,000 words from Reuters news articles. Each word is manually annotated with the named entity class for that word. The dataset is of high quality and freely available.

FAQ Sample dataset: Contains around 800 words from the SJSU CS department FAQ section. The dataset was created by manually annotating each word in the data sample.

A supervised learning based classifier allows us to train it on a domain that best fits the use case. Multiple models can be trained and ensembled to produce higher order classifiers. For example, we train the classifier on FAQ data for our middleware application and train it on the Wiki Gold dataset for a more general domain. For example, if we wanted to classify natural language text occurring in medical documents; we would train on a corpus of manually annotated medical text. The performance of a classifier depends on the quality, quantity, and domain of training data as well as the algorithm used for the model.

Before creating our classifier, we investigate the Named Entity Chunker (NEC) from the Natural Language Toolkit (NLTK) Python library. NEC is based on a model which implements a supervised machine learning algorithm called Maximum Entropy (ME). Maximum entropy is a very flexible method of statistical modeling which turns on the notion of "futures," "histories," and "features." Futures are defined as the possible outputs of the model. A "history" in maximum entropy is all the conditioning data which enables you to assign probabilities to the space of futures [15]. NEC comes pre-trained on the ACE corpus which is not freely available. As the classifier is serialized in the form a Python pickle file, it cannot be trained on a different training set. This makes NEC inflexible regarding being able to re-train the model on different domains depending on the use case. It is possible to inspect a model trained by the NEC and observe the most important features and their relative weights. We can also obtain the feature weights for all potential classifications and compare them to know why a classification was picked by the classifier. We create two functions that allow us to print this information in a readable form. Figure 2 shows why the word “California” was classified as a location (GPE) in the sentence, “STAR Act, is a California law designed to improve the interface between community college programs and CSU degree programs.” The output only displays the top 4 candidate classes.

```
ne_report('STAR act, is a California law designed to improve the interface between community college programs and CSU c
```

Explanation on the why the word 'California' was tagged:

Feature	B-GPE	B-ORGAN	O	B-GSP
prevtag=='O' (1)	3.767			
shape=='upcase' (1)	2.701			
pos+prevtag=='NNP+O' (1)	2.254			
en-wordlist==False (1)	2.095			
label is 'B-GPE' (1)	-2.005			
bias==True (1)	-1.975			
suffix3=='nia' (1)	1.700			
prefix3=='cal' (1)	1.139			
pos=='NNP' (1)	0.681			
prevword=='a' (1)	0.641			
nextpos=='nn' (1)	0.597			
word=='California' (1)	0.556			
wordlen==10 (1)	-0.399			
prevpos=='DT' (1)	-0.181			
word+nextpos=='california+nn' (1)	0.180			
nextword=='law' (1)	0.042			
prevtag=='O' (1)		3.389		
pos+prevtag=='NNP+O' (1)		1.725		
prevword=='a' (1)		1.093		
bias==True (1)		0.955		
en-wordlist==False (1)		0.837		
label is 'B-ORGANIZATION' (1)		0.718		
wordlen==10 (1)		0.605		
prevpos=='DT' (1)		-0.494		
prefix3=='cal' (1)		-0.474		
suffix3=='nia' (1)		-0.367		
word=='California' (1)		-0.278		
nextpos=='nn' (1)		-0.214		
pos=='NNP' (1)		0.174		
shape=='upcase' (1)		-0.084		
nextword=='law' (1)		-0.013		
bias==True (1)			10.125	
prevtag=='O' (1)			5.628	
shape=='upcase' (1)			-4.740	
label is 'O' (1)			-1.075	
pos=='NNP' (1)			-1.024	
en-wordlist==False (1)			0.698	
prefix3=='cal' (1)			-0.642	
prevpos=='DT' (1)			0.585	
wordlen==10 (1)			-0.556	
suffix3=='nia' (1)			-0.528	
nextpos=='nn' (1)			-0.488	
prevword=='a' (1)			-0.360	
nextword=='law' (1)			-0.199	
word=='California' (1)			-0.146	
pos+prevtag=='NNP+O' (1)			0.011	
prevtag=='O' (1)				2.925
pos+prevtag=='NNP+O' (1)				2.213
shape=='upcase' (1)				0.929
en-wordlist==False (1)				0.891
wordlen==10 (1)				-0.749
bias==True (1)				-0.592
label is 'B-GSP' (1)				-0.565
prevword=='a' (1)				0.487
pos=='NNP' (1)				0.393
nextpos=='nn' (1)				0.350
prevpos=='DT' (1)				-0.299
prefix3=='cal' (1)				0.219
suffix3=='nia' (1)				0.173
nextword=='law' (1)				0.066
word=='California' (1)				0.052
TOTAL:	11.793	7.570	7.289	6.495
PROBS:	0.855	0.046	0.038	0.022

Fig, 2. Top Four Potential Labels.

We test NEC using the Wiki Gold and FAQ datasets with accuracy as the primary scoring metric. This constitutes our baseline. The idea is to create a versatile classifier that uses a similar feature set as NLTK NEC but which can be trained on a large amount of training data from any domain. Logistic regression is a probabilistic model for binomial cases. It has been proven that maximum entropy generalizes the same principle for multinomial cases [16]. In both models, we want a conditional probability: $p(y|x)$ where y is the class and x is the vector of features. Logistic regression follows a binomial distribution whereas MaxEnt model uses the same principle but following a multinomial distribution. The sigmoid function is assumed in the derivation for logistic regression whereas, maximum entropy is assumed in the derivation of MaxEnt, and the sigmoid function is derived [17]. Our MaxEnt Classifier uses the following features for each word:

1. **Shape:** The shape of the word (e.g., does it contain numbers? does it begin with a capital letter?).
2. **Wordlen:** The length of the word.
3. **Prefix3:** The first three letters of the word.
4. **Suffix3:** The last three letters of the word.
5. **Pos:** The POS tag of the word.
6. **Word:** The word itself.
7. **En-Wordlist:** Does the word exist in an English dictionary?

8. **Prevpos:** The POS tag of the preceding word.
9. **Nextpos:** The POS tag of the following word.
10. **Prevword:** The word that precedes this word.
11. **Nextword:** The word that follows this word.
12. **Word+nextpos:** The word combined with the POS tag of the following word.
13. **Word+prevpos:** The POS tag of the word coupled with the tag of the preceding word.
14. **Prevshape:** The shape of the previous word.

To train the model, we need to extract values for these 14 features as well as the labels representing the named entity class for each word. Our feature extractor expects words with parts of speech tags and outputs a dictionary with the names of the 14 features as keys with corresponding values, for every word. We already know the labels for words in our training set. The feature dictionary for each word is combined with the label for that word in the form of a tuple and input to the classifier for training. At its core, the MaxEnt classifier uses the *LogisticRegression* class from the Python SciKit Learn library with a multinomial distribution. Figure 3 shows how we divide the data from the Wiki Gold dataset into training and test sets, instantiate the classifier, train it, and evaluate the model

based on its accuracy in predicting labels in the test data. This process is repeated for the FAQ dataset. The entire source code for classifier construction is available at [18].

```
# split the dataset into 75% training and 25% testing sets
def split_train_test(l, n):
    return [x[1] for x in enumerate(l) if x[0] % n], l[::n]

train_feats, test_feats = split_train_test(label_feats, 5)
len(train_feats), len(test_feats)
```

```
(31321, 7831)
```

```
from nltk.classify.scikitlearn import SklearnClassifier
from sklearn.linear_model import LogisticRegression
sk_classifier = SklearnClassifier(LogisticRegression(multi_class='multinomial', solver='lbfgs'))
```

```
# train maximum entropy classifier
start_time = time()
sk_classifier.train(train_feats)
print('Time to train: {:0.2f} sec'.format(time() - start_time))
print(sk_classifier)
```

```
Time to train: 3.52 sec
<SklearnClassifier(LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='multinomial',
          n_jobs=1, penalty='l2', random_state=None, solver='lbfgs',
          tol=0.0001, verbose=0, warm_start=False))>
```

```
# check accuracy of model using testing set
accuracy_model(sk_classifier, test_feats)
```

```
0.9337249393436343
```

Fig. 3. Classifier Training

The *accuracy_model* function uses the trained model to predict labels for the test set and compares the predictions to the actual classes. We thus use accuracy as our primary scoring metric. Scores for precision, recall, and f-measure are also calculated, as seen in figure 4.

```
# classify using model and verify accuracy score
start_time = time()
pred_list = make_predictions(test_feats)
true_labels = [label for feat,label in test_feats]
print('Time to classify: {:0.2f} sec'.format(time() - start_time))
print('Accuracy score: {}'.format(accuracy_score(true_labels, pred_list)))
```

```
Time to classify: 3.36 sec
Accuracy score: 0.9337249393436343
```

```
# evaluate using precision, recall and f1 scores
print('Precision score: {}'.format(precision_score(true_labels, pred_list, average='weighted')))
print('Recall score: {}'.format(recall_score(true_labels, pred_list, average='weighted')))
print('F1 score: {}'.format(f1_score(true_labels, pred_list, average='weighted')))

print('Confusion Matrix: \n{}'.format(confusion_matrix(true_labels, pred_list)))
```

```
Precision score: 0.9296451609262882
Recall score: 0.9337249393436343
F1 score: 0.930873141898728
```

Fig. 4. Classifier Evaluation

The data flow for predicting NE labels with the MaxEnt Classifier can be described as:

1. raw_input <= input (natural language source)
2. input_text <= normalize (raw_input)
3. input_tokens <= tokenize (input_text)
4. input_tags <= part_of_speech_tag (input_tokens)
5. feature_dictionaries <= extract_features(input_tags)
6. labelled_named_entities <= make_predictions (feature_dictionaries)

As seen in figure 5, for the models trained on the Wiki Gold training set, MaxEnt obtains a better accuracy score than the baseline. For the models trained on the FAQ Sample training set, MaxEnt obtains the same score as the baseline. It is possible that due to the relatively

small size of the FAQ Sample dataset, both classifiers perform similarly. For the much larger Wiki Gold dataset, however, the MaxEnt classifier performs significantly better.

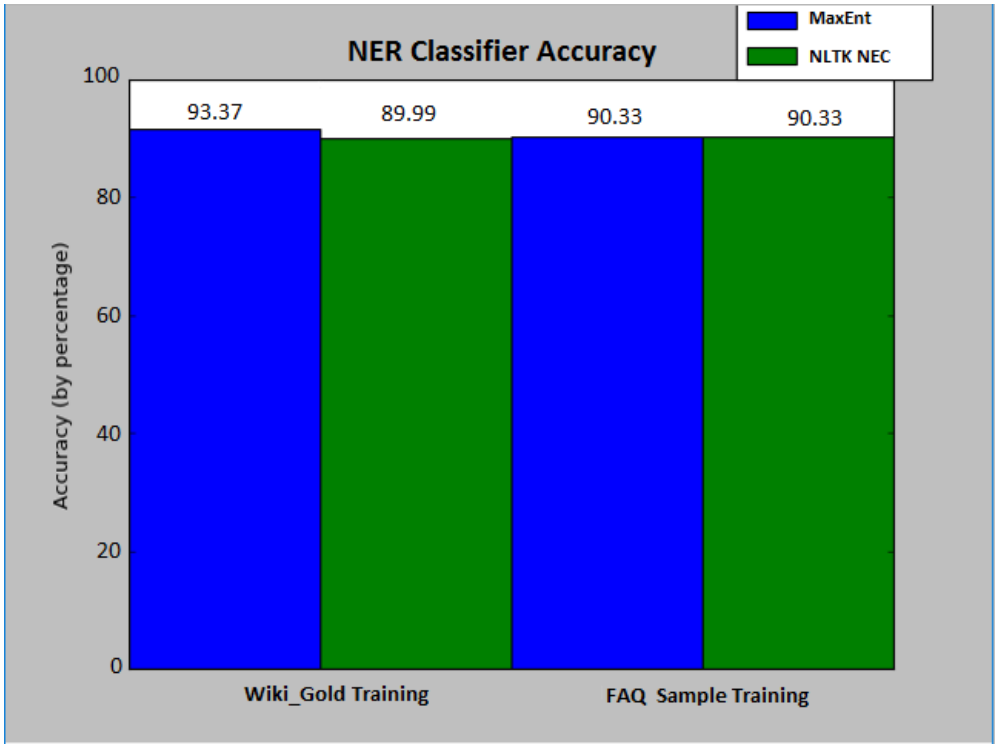

Fig. 5. MaxEnt vs NEC (baseline)

We persist the trained MaxEnt classifier in the form of a serialized Python pickle object so that we can use it in our implementations. All we need to do is load the pickled classifier and use it to make predictions on incoming data.

B. ChatBot Middleware

Before moving on to dealing with scalability, we implement the MaxEnt classifier in an example application. We build a chat bot application for the Slack messaging platform using BotKit, a JavaScript framework. Chat bots are programs that can communicate with human users or other chat bots, via various interfaces. For our use case, a chat bot using our classifier can identify the named entities present in a question asked by a student. The demonstration currently only prints the named entity labels for each word in a sentence, but this can be fleshed out by mapping certain actions to the outputs. The bot can then take actions depending on what type of named entities were present in the question. As described in the official documentation, BotKit allows developers to extend its functionality by creating middleware plugins. These middleware modules can processes messages to and from the core bot at several useful places. We wrap our classifier implementation into a REST API using Flask, a Python micro-framework and expose an HTTP endpoint for incoming requests as shown in figure 6. Flask is lightweight and allows us to specify multiple API endpoints easily using Python decorators.

```python
from logging import DEBUG

from flask import Flask, render_template, request, jsonify

from classify import classify_named_entities


app = Flask(__name__)
app.logger.setLevel(DEBUG)


@app.route('/')
@app.route('/index')
def index():
    return render_template('index.html')


@app.route('/classify')
def classify():
    if 'text' in request.args:
        predictions = classify_named_entities(request.args['text'])
        data = dict()
        for tup in predictions:
            data[tup[0]] = tup[1]

        resp = jsonify(data)
        resp.status_code = 200
        return resp

    return jsonify(dict())


if __name__ == '__main__':
    app.run(debug=True)
```

Fig. 6. REST API Implementation

We write a JavaScript module that acts as the middleware entry point and sends incoming messages to the exposed "/classify" REST API endpoint with a parameter "text," containing the message. The *classify_named_entities* function processes the input text, internally using

our classifier, which outputs the named entity classifications for each word in the sentence. As seen in figure 7, we get the named entities from the input text with their classifications as a response from our middleware.

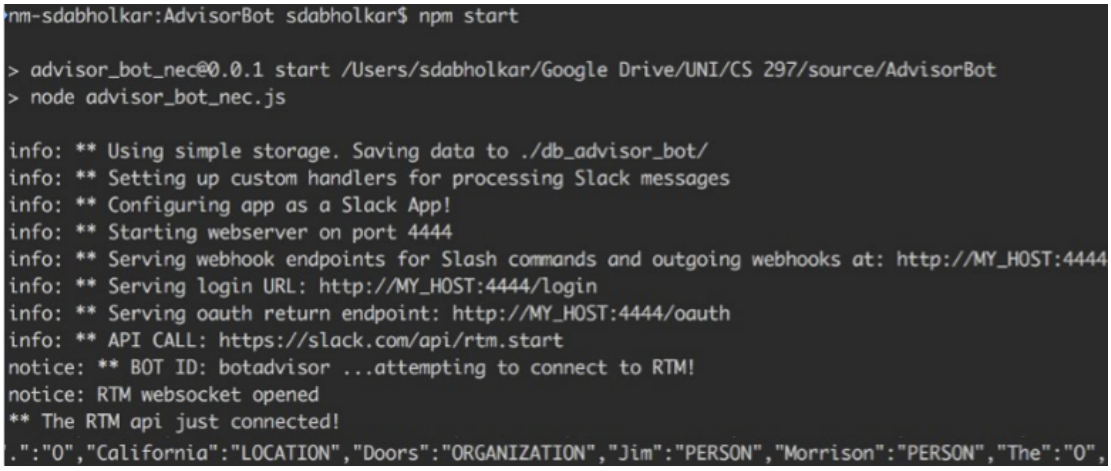

Fig. 7. Response from ChatBot Middleware

To assess scalability, we test running the middleware application for continuous concurrent requests. The performance for the concurrent scenario is compared to that of serial execution where there is only one request made every second, which will be the baseline. We use the lightweight Python load testing framework, Locust, to stress test the application. Figure 8 shows how Locust lets us define user behavior using python code. We simulate a user hitting our REST API "/classify" endpoint with the parameter text containing the sentence, "Jim Morrison was the lead singer of the band The Doors and lived

in California." We then use Locust to create a swarm of 100 clones with the help of the convenient web UI, who perform the same action every second.

```
from locust import HttpLocust, TaskSet


def login(l):
    l.client.post("/login", {"username": "ellen_key", "password": "education"})


def classify(l):
    l.client.get("/classify?text=Jim Morrison was the lead singer of the band"
                 + " The Doors and lived in California.")


def index(l):
    l.client.get("/")


def profile(l):
    l.client.get("/profile")


class UserBehavior(TaskSet):
    tasks = {index: 1, classify: 1}

    # def on_start(self):
    #     index(self)


class WebsiteUser(HttpLocust):
    task_set = UserBehavior
    min_wait = 0
    max_wait = 0
```

Fig. 8. User Behavior Defined Using Python Code

Tables I and II show the load testing results for the simulated scenarios with 1 user and 100 users respectively. It is observed that the average response time is much higher than for the later than the former. We also note that this value keeps increasing as the

application keeps receiving new requests. This is as expected because the application is single threaded at this point. Every new request is queued up and must wait for the current request being processed to complete.

TABLE I

Observations for 1 User Scenario

Method	Name	# requests	# failures	Median response time	Average response time	Min response time	Max response time	Average Content Size	Requests/s
GET	/classify?t	95	0	2200	2156	2030	2360	272	0.47
None	Total	95	0	2200	2156	2030	2360	272	0.47

TABLE II

Observations for 100 Users Scenario

Method	Name	# requests	# failures	Median response time	Average response time	Average response time	Max response time	Average Content Size	Requests/s
GET	/classify?t	12	0	13000	13922	13923	25605	272	0.51
None	Total	12	0	13000	13922	13923	25605	272	0.51

We could solve the problem of concurrent requests using multi-threaded or multi-processing server frameworks such as GUnicorn or CherryPy. However, the number of concurrent requests that the application can handle are still limited by the resources of a single machine. For a truly scalable solution, we must leverage a distributed computing framework.

C. Distributed Application

Figure 9 shows the architecture overview for our distributed implementation. The complete implementation comprises of multiple key components that allow us to deploy it at scale.

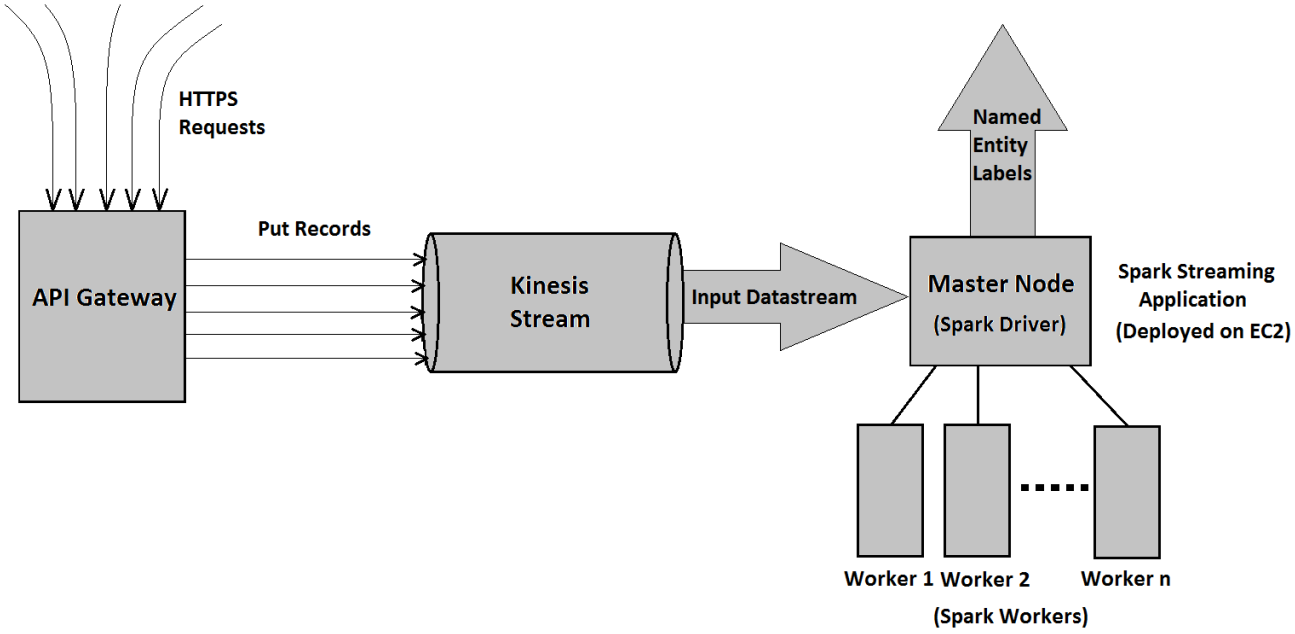

Fig. 9. Architecture Overview

1) API Gateway:

API Gateway makes it possible to create a RESTful API that can be used to interact with other services such as Kinesis, by exposing specific HTTPS endpoints. It lets us map these endpoints to commands for specific AWS resources.

2) *Kinesis:*

AWS Kinesis is a managed service that lets us read and write large amounts of data to and from it. Kinesis Streams supports the real-time ingestion of a significant amount of high-velocity data. A data record is the unit of data stored by the Streams service. A stream can be conceptualized as a chronological sequence of data records [19]. We refer to uniquely identified data records as a shard. A stream can have multiple shards. The number of shards in a stream can be increased or decreased as needed.

3) *Apache Spark:*

Apache Spark is a top level Apache project that provides a parallel processing framework for running large-scale data analytics applications across clustered computers. A Resilient Distributed Dataset (RDD), is the fundamental abstraction in Spark. It represents an immutable, partitioned collection of elements that can be operated on in parallel. As per the official documentation [20], "Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams." Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a

continuous stream of data. A DStream can be conceptualized as a continuous sequence of RDDs.

4) *EC2:*

Amazon Elastic Compute Cloud (EC2) provides scalable computing capacity in the AWS cloud. We can use EC2 to launch as many or as few virtual servers as we need, with the required software and hardware configurations. Such flexibility enables us to scale up or down to handle changes in requirements without the trouble of managing and setting up expensive hardware.

We direct incoming requests into an AWS Kinesis stream via an API Gateway HTTPS endpoint. A consumer continuously reads from the stream and serves the data as a text stream. We design and implement a Spark Streaming application which takes this text stream as input and outputs the named entity classifications for each word in every sentence. We deploy the application on two AWS EC2 clusters, one with 3 worker nodes and the other with 9 workers. Application performance is observed and compared across the two clusters. As described in the Apache documentation [20], “Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.” Figure 10

shows the process of Spark divides the input stream into ordered micro-batches and computes each batch to produce output.

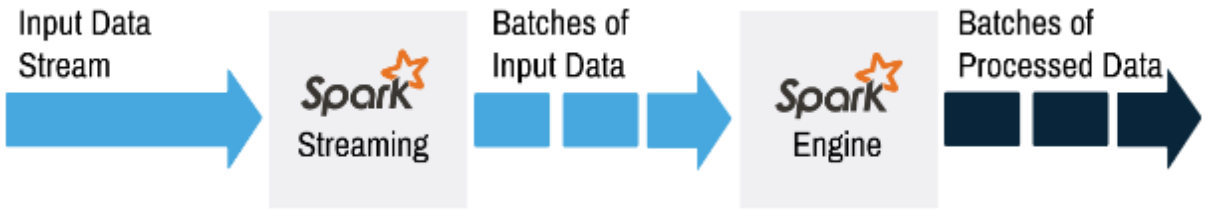

Fig. 10. Spark Micro-batches

We vary the window size (in seconds) for our application at different loads and observe its effects on throughput (records processed per second). A point to note is that making the window longer increases the response time of an application. Once a data stream is created, Spark lets us transform it by applying (using operations such as map) operations on each element in the stream. Spark evaluates directed acyclic graphs (DAG) for the data transformations lazily, i.e. it does not execute the transformations until it encounters an action such as *count()* or *pprint()*.

A Kinesis consumer program continuously reads the data from our Kinesis stream and serves it on a local port. The Spark driver program connects to the consumer via the local

port and gets the input request text data in the form of a data stream. Figure 11 shows how we create the text data stream and apply (or map) multiple data transformations on it to obtain the named entity labels. The second argument to the *StreamingContext* constructor in line 123 specifies the window length in seconds.

```
120 if __name__ == '__main__':
121
122     sc = SparkContext(appName='StreamingKinesisTestNEC')
123     ssc = StreamingContext(sc, 15)
124
125     textStream = ssc.socketTextStream('172.24.78.141', 9997)
126     neStream = textStream.map(tokenize)         \
127                          .map(pos_tag)          \
128                          .map(extract_features) \
129                          .map(make_preicttions)
130     neStream.pprint(999)
131     ssc.start()
132     ssc.awaitTermination()
```

Fig. 11. Spark Driver

A map operation applies a function specified as the argument, on each record of input. Spark distributes this execution amongst the worker nodes and it also handles the distribution of compute. The scheduling of jobs and the collection of outputs is done under the hood so we can focus on writing our application logic. In figure 11, the data flow transformations previously discussed are applied on the stream as seen in lines 125 to 129. Each record comprised of a single sentence is tokenized, tagged with its part of speech,

passed to the feature extractor to create a list of feature dictionaries and then to the classifier to predict named entity classes for each word in the sentence. If we want to map (apply) a function onto a data stream (or batch), the function must be a closure. A closure is a first-class function that also stores its environment. As seen in figure 12 for example, the *tokenize* and *pos_tag* functions we use, contain the imports they need for execution inside the function definition. The function is shipped to the worker nodes for execution and any imports outside this closure will not be valid. An implication of this is that the library or module the function imports, must be available on the worker nodes. Thus, the function is executed where the data is located (worker in-memory) and data locality is leveraged.

```
5  def tokenize(text):
6      from nltk.tokenize import WordPunctTokenizer
7      tokens = WordPunctTokenizer().tokenize(text)
8      return tokens
9
10
11 def pos_tag(tokens):
12     from nltk import pos_tag
13     return pos_tag(tokens)
```

Fig. 12. Closure Example

We similarly create closures for our feature extractor and prediction functions. Figure 13 shows the *make_predictions* function. It loads the pickled classifier and uses it to make label

predictions from the features of every word. The function returns a dictionary containing each word in a sentence and its corresponding classification.

```
95  def make_preicttions(tokens_feats):
96
97      import pickle
98      with open('/home/ubuntu/wiki_gold_me_classifier.p',
99                'rb') as pkl:
100         me_classifier = pickle.load(pkl)
101
102     tokens = list()
103     feat_dicts = list()
104     preds = list()
105     tokens_preds = list()
106
107     for tagged_token, feat in tokens_feats:
108         tokens.append(tagged_token[0])
109         feat_dicts.append(feat)
110
111     for feat_dict in feat_dicts:
112             preds.append(me_classifier.classify(feat_dict))
113
114     for token, pred in zip(tokens, preds):
115         tokens_preds.append((token, pred))
116
117     return tokens_preds
```

Fig. 13. Prediction Function

When a Spark streaming application is running, various information and statistics related to execution are available on the Spark web UI. By default, the web UI can be found on port 4040 of the node running the driver program. We can view graphs such as those for input rate, average processing time, scheduling delay and the same details for each micro-batch execution. We collect this information for each run and use it to calculate aggregates.

VI. EXPERIMENTS

We test the performance of the distributed application by simulating three different load scenarios for two clusters, one with three workers and the other with nine workers.

i. *Low*: With approximately 10 requests per second on average.

ii. *Medium*: With approximately 50 requests per second on average.

iii. *High*: With approximately 500 records per second on average.

We also run the application with an input rate of 1 request per second to simulate a non-concurrent scenario, which serves as our baseline. We use a Kinesis producer script we wrote using Python, to continuously put randomly generated sentences into a Kinesis stream, simulating incoming requests. We run the producer in parallel, using multi-processing to create the required input loads.

Running the application at different loads, we observe the effects of modifying the window length for each. We define throughput as records processed per second, where each record corresponds to one request, and use it as our scoring metric. The values for parameters such as processing time, window length, micro-batch size and total delay are noted for each

run. The experiment is performed identically on the two clusters. We begin by fixing the window length to 5 seconds and running the application with an input rate of 1 request per second (our baseline). We then execute the application with an input rate of around ten requests every second and compare with the baseline. This step allows us to test if our implementation was successful in speeding up the process of handling concurrent requests. We then run the application for medium and high loads. For each load, we observe the scores obtained for multiple executions, with increasing window lengths (seconds) for every run.

We aggregate the data collected for every batch execution for a given run, for all runs. Thus, we effectively condense observations form each execution into a row, and tabulate it into a single table for that cluster. We use the two final tables (one for each cluster) thus obtained, to study the observations and make performance comparisons.

VII. RESULTS

The observations from the various runs are listed in Tables III and IV. Window size represents the number of seconds for which each batch is filled with records. Input size indicates the average number of records in each batch. Processing time is the average time taken to process each batch. Total delay is the average duration a batch had to wait from its creation to it being processed and includes the scheduling delay for a batch. Throughput represents the average number of records processed every second for each run. We load the observations data into Tableau and visualize it to help us compare and interpret the results

TABLE III

Observations for Cluster with Three Workers

Load	Window (sec)	Input Size (records)	Processing Time (sec)	Total Delay (sec)	Throughput (recs/sec)
one	5	4.90	0.881	0.881	5.5676
low	5	61.11	6.778	13.556	9.0164
med	5	244.11	26.111	113.778	9.3489
med	10	443.30	39.800	157.700	11.1382
med	15	739.21	58.316	465.474	12.6760
med	20	989.71	74.471	552.706	13.2899
med	25	1,230.67	85.333	346.000	14.4219
med	30	1,496.27	90.545	418.364	16.5251
high	5	2,217.75	189.000	462.000	11.7341
med	120	5,795.17	86.000	86.000	67.3857
high	15	6,359.75	183.000	595.500	34.7527
high	30	12,640.50	175.500	460.500	72.0256

TABLE IV

Observations for Cluster with Nine Workers

Load	Window (sec)	Input Size (records)	Processing Time (...	Total Delay (sec)	Throughput (recs/sec)
one	1	0.95	0.325	0.425	2.923
med	5	241.00	20.667	81.556	11.661
med	20	945.10	38.500	116.900	24.548
med	30	1,299.10	41.500	93.100	31.304
med	120	5,907.25	50.250	50.250	117.557
high	15	7,730.75	184.500	477.000	41.901
high	30	14,864.88	193.500	730.500	76.821

Figure 14 shows the comparison of loads of one request per second and 10 (low load) requests per second respectively. For a fixed window length of 5 seconds, we observe that it takes 0.881 seconds on average to process a batch of roughly 5 requests. Thus, the throughput is 5.56 recs per second for our baseline. On the other hand, it takes 6.778 seconds on average to process a batch of roughly 60 requests. Thus, the throughput is 9.01 recs per second, an improvement over our baseline.

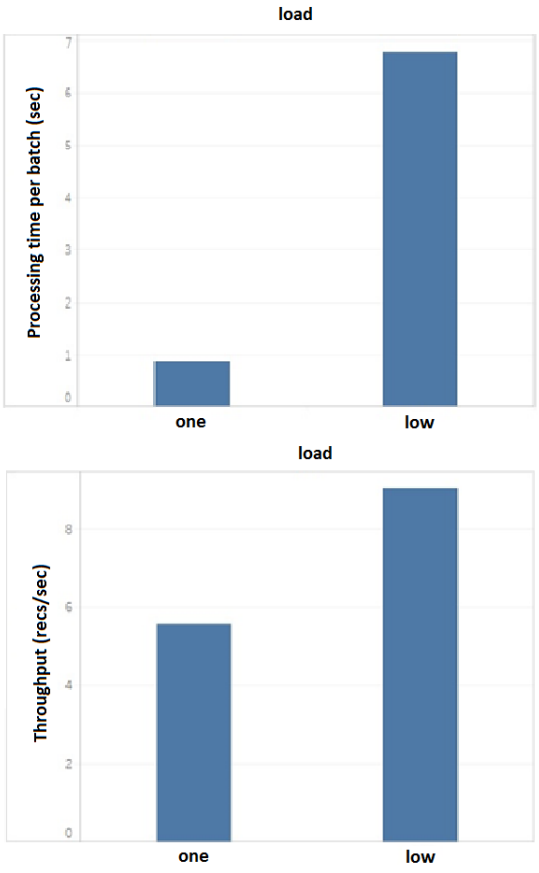

Fig. 14. Baseline vs. Low Load Performance

From this, we infer that our implementation can handle concurrent requests more efficiently. Figure 15 compares the observations for medium and high load runs and increasing window lengths, for the cluster with three workers.

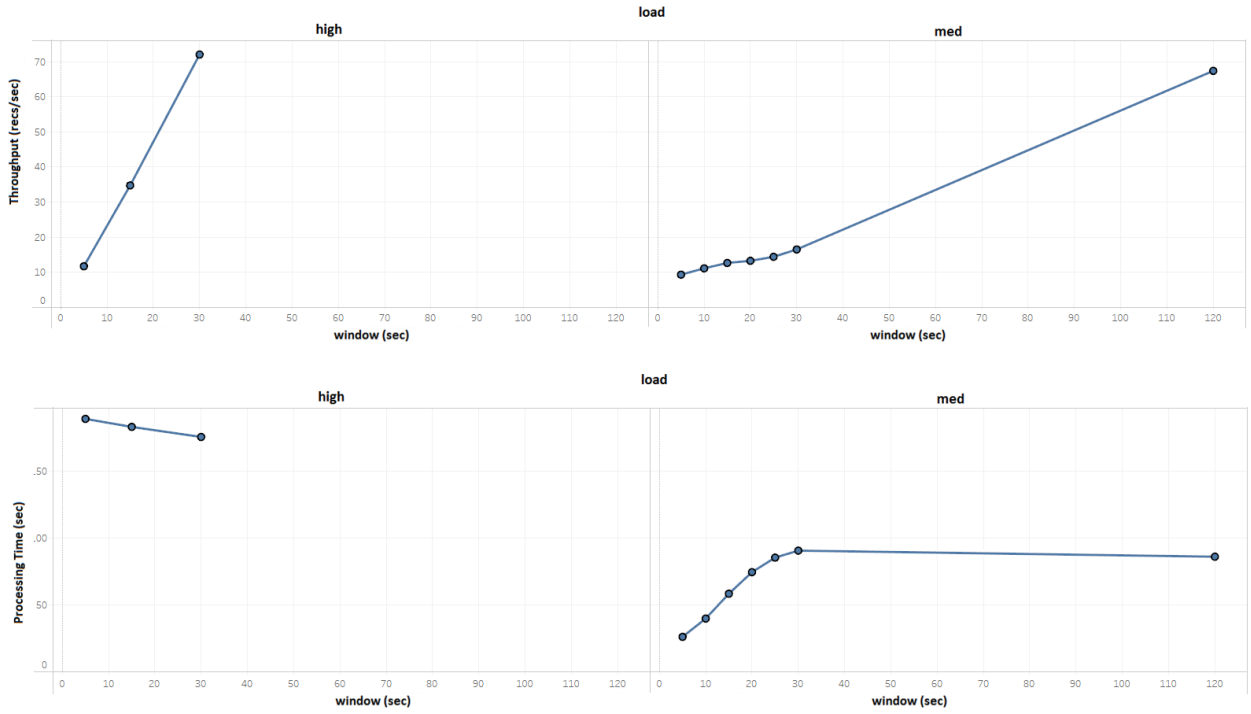

Fig. 15. Medium vs High Load Performance for 3 Worker Cluster

We observe that initially, as we increase the window length, the time taken to process a batch increases. As we increase the window length, more records are collected for every batch, increasing the input batch size. The results indicate that the increase in processing time is not directly proportional to the increase of input batch size. A rising throughput

suggests that with an increase in input batch size, the time taken to process a single record decreases i.e. the system performs more efficiently. After a point, the processing time roughly evens out with an increasing window length. We observe the highest throughput scores in this range. Running the experiment again on a larger cluster allows us to verify our observations as well as observe the effects of additional execution cores on performance. Figure 16 compares the observations for medium and high loads and increasing window lengths for the cluster with nine workers. Like the smaller cluster, we observe that increasing the window size for each micro-batch boosts the number of records processed per second. We infer from the two sets of observations that the performance of our application varies depending on the size of each input batch. Creating larger batches by increasing the window length is observed to increase throughput.

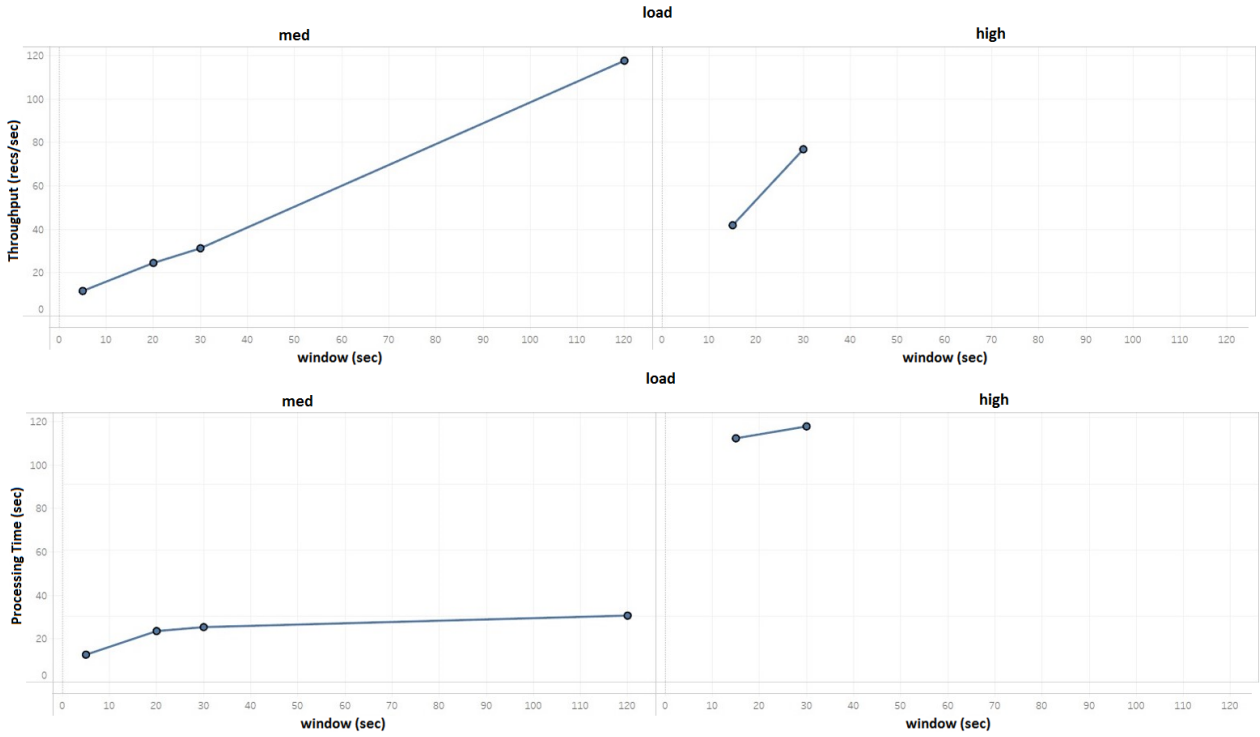

Fig. 16. Medium vs High Load Performance for 9 Worker Cluster

From a practical perspective, however, it may not be feasible to increase the window length beyond a point. In the case of our middleware implementation, for example, increasing the window length also increases the response time as every request in a batch must wait longer for the batch to be filled. Desired response times will depend on the use case. Owing to the nature of the chatbot middleware application, we want to minimize response time and maximize throughput. To achieve this, we would want an input batch size that gives us a high enough throughput for the smallest window length to ensure as short a response

time as possible. Figure 17 shows the rise in throughput per the increase in input batch size.

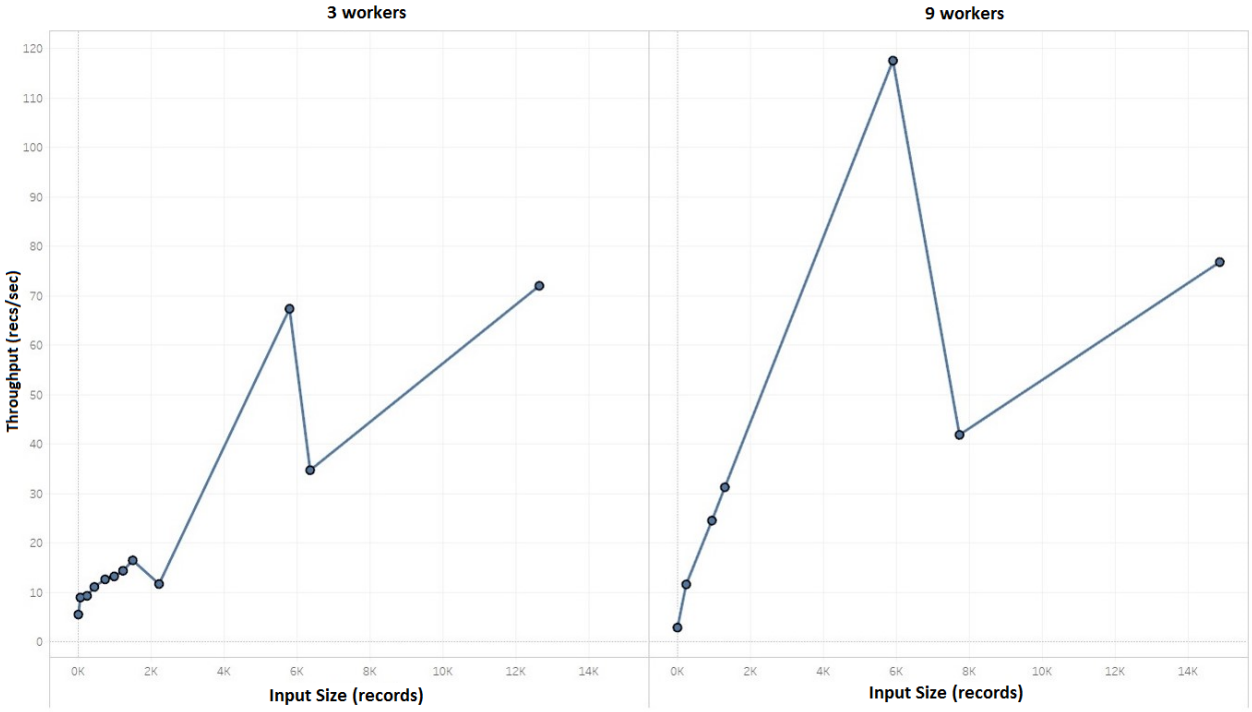

Fig. 17. Effect of Input Size on Throughput

We observe a similar trend in the observations for the two clusters. A local maximum is observed for an input size range between 5795 to 5910 records per batch at medium loads and a window length of 120 seconds. This duration is an acceptable length for our example use case. We can, therefore, tune our application for these optimal parameters.

VIII. CONCLUSION

Named Entity Recognition and Classification (NERC) is a field in which much research has been done and yet, it is still a challenging problem. Many factors affect the performance of NERC classifiers such as the type of classifier implemented, quality, quantity and domain of datasets as well as the scale of the use case. By means of this project, we explored recent research done in the field of NERC and built a supervised learning classifier based on the maximum entropy model. Implementing the classifier in a chatbot application allowed us to demonstrate a real-world use case, as well as load test the implementation for concurrent input requests. We observed that the performance for a higher number of concurrent requests is slow and limited by the resources of a single machine. Implementing the classifier as part of a distributed computing framework, prototyping and deploying it on two clusters of different sizes allowed us to handle very high-velocity simultaneous inputs at scale. We tested our solution with various loads using our example use case for context. We compared the observations for all runs and found an optimal configuration that is acceptable for our use case.

Big data is characterized by high velocity, volume and variety. We have so far focused on high velocity data. We can work towards making the application scale for very large size

inputs. For our ChatBot application, certain actions can be mapped to specific named entities recognized from text. For example, on recognizing that a word represents a person's name, our bot can look up a database to check if it is a CS department faculty and provide the relevant contact information to the student asking the question.

IX. REFERENCES

[1] S. Bird, "NLTK: the natural language toolkit," in *Proceedings of the COLING/ACL on Interactive presentation sessions, Association for Computational Linguistics*, 2006. [Online]. Available: http://delivery.acm.org/10.1145/1230000/1225421/p69-bird.pdf?ip=24.130.50.198&id=1225421&acc=OPEN&key=4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E6D218144511F3437&CFID=878818112&CFTOKEN=42448333&__acm__=1482284911_829169a1dc503d852d433f8e411b64ff]

[2] L. Ratinov and D. Roth, “Design challenges and misconceptions in named entity recognition,” in *Proceedings of the Thirteenth Conference on Computational Natural Language Learning - CoNLL '09*, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1596399

[3] R. Jiang, R. E. Banchs, and H. Li, “Evaluating and combining Name Entity Recognition Systems,” in *Proceedings of the Sixth Named Entity Workshop*, 2016. [Online]. Available: http://www.aclweb.org/anthology/W/W16/W16-27.pdf#page=31

[4] T. K. Sang and F. Meulder, “Introduction to the CoNLL-2003 shared task,” in *Proceedings of the seventh conference on Natural language learning at HLT-NAACL*, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=1119195

[5] D. Nadeau and S. Sekine, “A survey of named entity recognition and classification,” in *Current Topics Named Entities Recognition: Classification and Use*, 2009, pp. 3–28. [Online]. Available: http://www.ingentaconnect.com/content/jbp/li/2007/00000030/00000001/

[6] R. Grishman, B. Sundheim, “Message Understanding Conference - 6: A Brief History,” in *Proceedings of the International Conference on Computational Linguistics*, 1996. [Online]. Available: http://www.alta.asn.au/events/altss_w2003_proc/altss/courses/molla/C96-1079.pdf

[7] Coates-Stephens, “The Analysis and Acquisition of Proper Names for the Understanding of Free Text,” in *Computers and the Humanities vol. 26*, Morgan Kaufmann Publishers, San Francisco, CA, 1992, pp.441-456.

[8] T. Poibeau, L. Kosseim, “Proper Name Extraction from Non-Journalistic Texts,” in *Proceedings Computational Linguistics,* Netherlands, 2001, vol. 37, pp. 144-157. [Online]. Available: http://www.ingentaconnect.com/content/rodopi/lang/2001

[9] S. Sekine, C. Nobata, “Definition, Dictionaries and Tagger for Extended Named Entity Hierarchy,” in *Proceedings of the Conference on Language Resources and Evaluation*, 2004.

[10] L. Chiticariu, R. Krishnamurthy, Y. Li, F. Reiss, S. Vijayanathan, “Domain Adaptation of Rule-Based Annotators for Named-Entity Recognition Tasks” in *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, 2010, pp. 1002-

1012. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.173.697&rep=rep1&type=pdf

[11] E. Riloff, R. Jones, “Learning Dictionaries for Information Extraction using Multi-Level Bootstrapping,” in *Proceedings of the National Conference on Artificial Intelligence*, 1999, pp. 474-479. [Online]. Available: http://www.aaai.org/Papers/AAAI/1999/AAAI99-068.pdf

[12] R. Giuseppe, R. Troncy, "Nerd: evaluating named entity recognition tools in the web of data," in *Workshop on Web Scale Knowledge Extraction Bonn*, Germany, 2011, pp. 1-16. [Online]. Available: http://porto.polito.it/2440793/1/wekex2011_submission_6.pdf

[13] E. Alfonseca, S. Manandhar, “An Unsupervised Method for General Named Entity Recognition and Automated Concept Discovery,” in *Proceedings of the International Conference on General WordNet*, 2002, pp. 34-43. [Online]. Available: http://www4.ncsu.edu/~mbcusick/papers/alfonseca2002unsupervised.pdf

[14] Y. Shinyama, S. Sekine, “Named Entity Discovery Using Comparable News Articles,” in *Proceedings of the International Conference on Computational Linguistics*, 2004, pp.848. [Online]. Available: http://dl.acm.org/citation.cfm?id=1220477

[15] I. Ahmed, R. Sathyaraj, "Named Entity Recognition by Using Maximum Entropy," in *International Journal of Database Theory and Application 8.2*, 2015, pp. 43-50. [Online].

Available:
https://pdfs.semanticscholar.org/317a/5c881c940c9b5beb706bdac1bec52fc528f3.pdf

[16] D. Klein, C. Manning, "Maxent Models, Conditional Estimation, and Optimization without Magic," in *The 41st Annual Meeting of ACL*, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=1075176

[17] J. Mount, "The equivalence of logistic regression and maximum entropy models," 2011. [Online]. Available: http://www. win-vector.com/dfiles/LogisticRegressionMaxEnt. pdf

[18] S. Dabholkar, “Named Entity Classifier,” GitHub Page. [Online]. Available: https://github.com/Shreeraj1746/Named_Entity_Classifier/blob/master/MaxEnt_NEC.ipynb

[19] Amazon Kinesis Streams Key Concepts, Developer Documentation. [Online]. Available: http://docs.aws.amazon.com/streams/latest/dev/key-concepts.html

[20] Spark Streaming Programming Guide Documentation. [Online]. Available: http://spark.apache.org/docs/latest/streaming-programming-guide.html