San Jose State University
SJSU ScholarWorks

Master's Projects

Master's Theses and Graduate Research

Spring 6-23-2017

Question Type Recognition Using Natural Language Input

Aishwarya Soni
San Jose State University

Recommended Citation

Soni, Aishwarya, "Question Type Recognition Using Natural Language Input" (2017). *Master's Projects*. 542.
DOI: https://doi.org/10.31979/etd.8xd4-79bk
https://scholarworks.sjsu.edu/etd_projects/542

Question Type Recognition Using Natural Language Input

A Writing Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

By

Aishwarya Soni

May 2017

The designated Project Committee Approves the Project Titled

Question Type Recognition Using Natural Language Input

By

Aishwarya Soni

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2017

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

Dr. Robert Chun (Department of Computer Science)

Prof. James Casaletto (Department of Computer Science)

ABSTRACT

Recently, numerous specialists are concentrating on the utilization of Natural Language Processing (NLP) systems in various domains, for example, data extraction and content mining. One of the difficulties with these innovations is building up a precise Question and Answering (QA) System. Question type recognition is the most significant task in a QA system, for example, chat bots. Organization such as National Institute of Standards (NIST) hosts a conference series called as Text REtrieval Conference (TREC) series which keeps a competition every year to encourage and improve the technique of information retrieval from a large corpus of text. When a user asks a question, he/she expects a correct form of answer in reply. The undertaking of classifying a question type is to anticipate the sort of a question which is composed in common dialect. The question is then classified to one of the predefined question types. The objective of this project is to build a question type recognition system using big data and machine learning techniques. The system will comprise of a supervised learning model that will receive a question in a natural language input and it can recognize and classify a given question based upon its question type. Extracting important textual features and building a model using those features is the most important task of this project. The training and testing data has been obtained from the TREC website. Training data comprises of a corpus of unique questions and the labels associated with it. The model is tested and evaluated using the testing data. This project also achieves the goal of making a scalable system using big data technologies.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Thanh Tran, for his continuous guidance, patience, motivation and enthusiasm throughout this project. His guidance helped me throughout this project, starting from topic selection and until writing my Masters project. He was always ready to help whenever I had a question and the discussion with him gave me the direction to how to approach the problem. The experience of working with him was very valuable and I really learnt a lot throughout this project.

I would like to thank my committee members Dr. Robert Chun and Prof. James Casaletto for their continuous support and valuable time to review my work and also guide me by providing valuable feedback whenever needed.

Finally, I would like to thank my parents, family members and my friends for their immense support and encouragement throughout my Masters program. This would not have been possible without them.

Table of Contents

1. Introduction

Currently, there is huge amount of information that is available on the web and the web crawlers are required to be more insightful. Much of the time the client just needs a particular snippet of data rather than a rundown of reports. Instead of making the client to peruse the whole report, it is regularly wanted to give the client a brief and short answer. The QA systems are designed to provide an accurate answer to a particular question. An ideal QA system is the one which can answer any question in a human readable format.

The study to create an accurate QA systems began in the year 1960 where the first QA system, BASEBALL was designed [1]. It was capable of answering domain specific questions, which in this case was the data of baseball games played in American league over one season. But the QA system was incapable of answering out of domain questions and that was the limitation that needed to be improved. As the years passed by, the data became huge on the web and there was a need for a system to retrieve the required information out of such a huge database. Thus, the need to develop a better QA system came into focus again. In 1999, TREC began the competition to create a better QA system. Participants in the TREC competition needed to build a model that can retrieve the precise answer, from a 10 GB of text corpus which comprises of English sentences.

Designing a precise question answering system has been always a challenging task. It is different from a document retrieval system where the documents are retrieved based upon some keywords or text matching.

The most challenging task in designing a QA system is to precisely find the intent behind a particular question. Natural language processing techniques were exploited to build a system which can process text in any human readable language. In order to correctly classify a question, the system needs to understand what is the intent behind the question asked, what are the constraints associated with it and what can be the possible answers to that question. For example, given a question, " *What two states is Washington D.C. between?"* the system needs to identify that the intent is related to some location.

This project is build on the similar concept. It comprises of using advance natural language processing methods along with machine learning technique. Different classification algorithms can be used to build a model which can process the textual features extracted using natural language tool kit.

Model is build using a supervised learning approach. In supervised learning we have the training data along with the labels and we build a model using this data. We later on evaluate the model using the testing data to check the accuracy on unknown sentences. The model is then scaled using Apache Spark to handle a large stream of requests from multiple users simultaneously. Project is build using several steps which includes data preprocessing, feature extraction, model training, prediction and testing and evaluating the performance which are all explained in the later sections.

2. Background

Multiple techniques were implemented since 1960, when the first question answering system was developed. But the drawback for those systems was that they were binded to a particular domain. During the era of 1970s and 1980s, the interest to use natural language processing techniques got a boost and the systems were proposed based on NLP methodologies. UC Berkley came up with a project, The Berkeley Unix Consultant project (UC) [2]. They build the project on the domain of Unix operating system and came up with a system that utilized the techniques such as natural language processing, knowledge representation system and reasoning. Based upon a question, the system generated a knowledge representation and the answer was tailored based upon the knowledge the system generated. But due to lack of support for real-world scenarios, the system was not considered to be used for practical purpose. Rule based systems were proposed build on the concept of knowledge representation systems, but due to increasing amount of data, the number of rules increased and it became very difficult to manage the system with too many rules.

Since the start of the QA track by TREC in 1999, the focus changed from domain specific systems to Information Retrieval (IR) systems. The most simple QA system was based on answering *factoid questions,* for example, "*Who was the first person to land on the Moon?"* This is a factual question which has the precise answer. The advent of IR systems gave birth to open-domain question answering system [2]. This is the current state of the art on which multiple automated question answering systems are build.

3. Related Work

Even though there are multiple papers and works that are proposed, I am extending the work done by Xin Li and Dan Roth [3] while building the solution. They came up with a system which was based upon the concept of open-domain answering system. The model that they proposed was build using machine learning and natural language processing techniques. The model was trained using the train_5500 dataset and evaluated using TREC_10 dataset. The features such as *words, pos_tags, head_chunks, named entities* were extracted and a model was fitted with those features. They achieved the accuracy of 85% with the proposed model.

3.1 Drawbacks of the system:

- Advanced Information Retrieval technique such as term frequency- inverse document frequency (tf-idf) was not used. TF-IDF helps in recognizing the importance of a particular word in the entire corpus of text. In this technique, the word is assigned a weighted frequency depending upon the number of times it appears in the entire corpus. It is the advanced text mining technique which is important to consider as some words appear more number of times as compared to others. According to Wikipedia, around 83% of the text based recommendation systems uses tf-idf.

- Understanding the semantics is also one of the important feature that the current question answering systems are build on. It is important to consider the semantics of a sentence as a single word can be represented in multiple ways. For example, the word "destination" in a sentence can be replaced by the word "last stop". The meaning of the sentence would still remain the same. It is important to understand the semantic nature of the sentence

- Ngram helps in understanding the semantics of a sentence. Ngram takes into consideration the sequence of words that occur continuously in a given sentence. The value of n defines the model. For example, if n = 2, it is called a bigram model and if n = 3, it is called as trigram. For the values of n > 3, it is generally alluded to four grams, five grams etc. For example, if we take n = 2, the bigram for the sentence, "*What is the currency of the USA?"* will be,
    - What is
    - is the
    - the currency
    - currency of
    - of the
    - the USA

  Ngrams are used as a feature in the text processing model. This is an important feature to consider as the sequence of words can identify which words are more frequently used to denote a person, or organization or location or any other entity. This improves the learning of the model which may help in improving the performance of classification.

- As there are lot many users to use a system, it was necessary to make a scalable system. Scalability was missing in the system that was implemented.

4. Proposed Solution

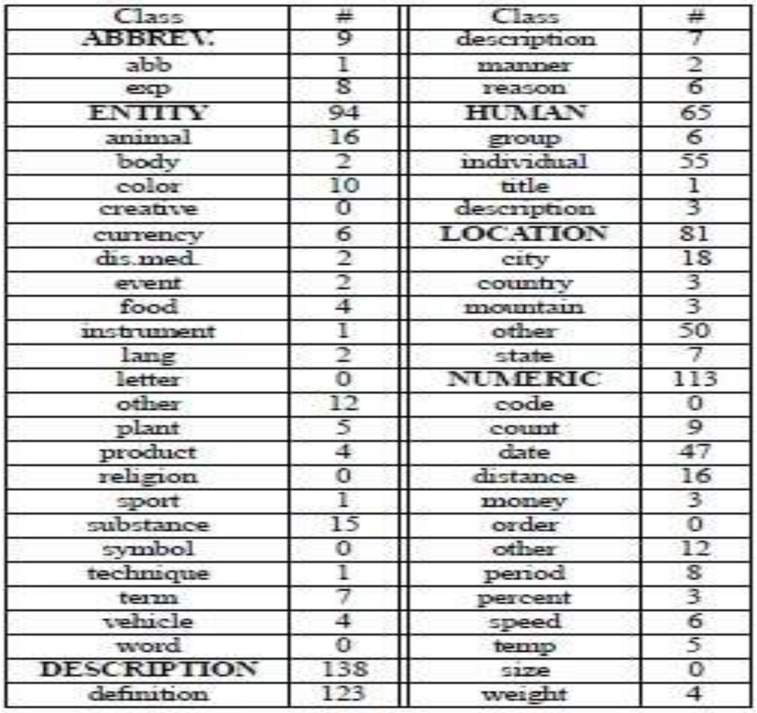
The solution proposed tends to build a more accurate question recognition system. It will incorporate the advance natural language processing techniques to create a better model. The proposed system will address the drawbacks of the previously build system. The system will provide a better accuracy and it will be tuned to support multiprocessing and scalability.

The system will receive a question in an input. It will process the text by extracting important features and finally the output will be a question type. The system will be fast, scalable, efficient and will be capable enough to handle bulk of request.

5. Data

The dataset has been obtained from the TREC website [4]. The training data train_5500 consists of 5500 labeled questions. The system is evaluated using TREC_10 dataset which consists of 500 new questions. The dataset consists of six broad categories of questions and fifty fine categories of questions. The output will be in the format, **BROADCATEGORY:fineCategory**. For example, the output of the question *"Who is Zebulon Pike?"* will be **"HUM:desc"**. Each question is uniquely classified to a particular category.

The following figure gives a distribution of different question categories. The numbers in the # column indicate the total number of questions available in the TREC_10 dataset in that particular category.



Class	#	Class	#
ABBREV.	9	description	7
abb	1	manner	2
exp	8	reason	6
ENTITY	94	HUMAN	65
animal	16	group	6
body	2	individual	55
color	10	title	1
creative	0	description	3
currency	6	LOCATION	81
dis.med.	2	city	18
event	2	country	3
food	4	mountain	3
instrument	1	other	50
lang	2	state	7
letter	0	NUMERIC	113
other	12	code	0
plant	5	count	9
product	4	date	47
religion	0	distance	16
sport	1	money	3
substance	15	order	0
symbol	0	other	12
technique	1	period	8
term	7	percent	3
vehicle	4	speed	6
word	0	temp	5
DESCRIPTION	138	size	0
definition	123	weight	4

Figure 1. Distribution of question categories

6. Architecture

The following diagram depicts the process flow for the proposed solution,

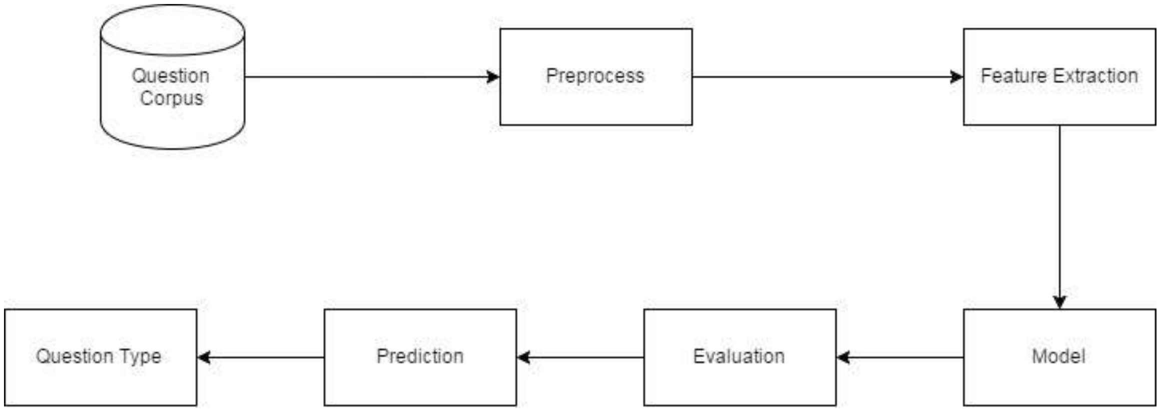


Figure 2. Architecture of Proposed Solution

The entire architecture can be explained as follows,

1. Question corpus is the training data, train_5500 obtained from the TREC.

2. The corpus is preprocessed by removing some unwanted characters and the processed corpus is then used to extract the features

3. Features such as Term Frequency- Inverse Document Frequency (TF-IDF), Named Entity Recognition (NER), Parts Of Speech (POS) tags, Related word vectorizer (for semantic analysis) are extracted and is given to the machine learning algorithm SVM to build the model.

4. SVM takes all the features, fits the training data.

5. The model is evaluated using TREC_10 testing dataset

6. The model is supplied with new sentence to make a prediction of a question type

7. The output of the model is a question type from the entire question type hierarchy as shown in Figure 1.

6.1 Rules for Classification

Following are the rules for labeling the data,

- If a query starts with *Who or Whom: type* ***Person.***
- If a query starts with *Where: type* ***Location.***
- If a query contains *Which or What, the head noun* phrase determines the class, as for What X questions.

6.2 Challenges while developing the model

- Precisely predict the *k semantic classes.*
- Using advanced NLP techniques to understand the semantics of a given question rather than just extracting a key term
- Make a scalable model
- Ability to handle load

7. Implementation Using Scikit Learn

Scikit-learn is a machine learning library in Python which is used for data analysis and data mining. It consists of various machine learning libraries which can be used for classification, regression or clustering. Question type recognition is a classification problem and so we are using a classification library which is Support Vector Machine (SVM).

7.1 Why SVM for text classification?

During the course of the years, researchers have tried using different algorithms combination for text classification. But Support Vector Machine (SVM) performs better than most of the algorithms. Thorsten Joachims have mentioned a few reasons why SVM outperforms the other algorithms [5],

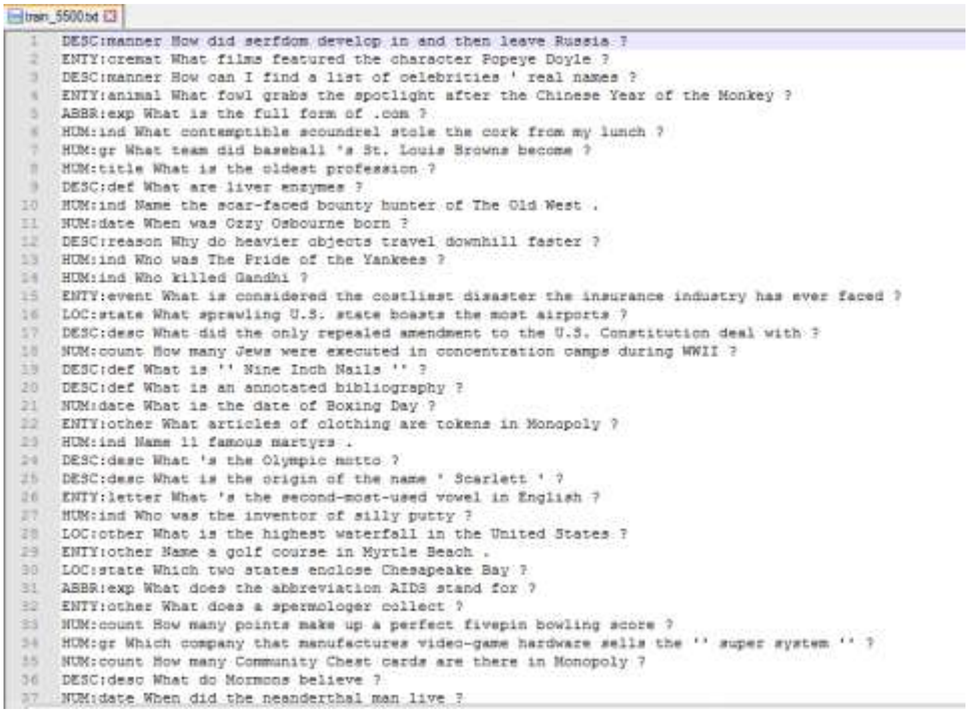
- SVM can handle large feature space. There are lot many features to handle while making a model for text classification, so there may be a chance of overfitting the data so as to cover all the features as close as possible. Since SVM can prevent overfitting with an appropriate regularization parameter (C), it does not have to rely on the quantity of features, they can possibly deal with large number feature spaces.
- There is no need of feature selection in text classification as most of the textual features are important. So unlike other types of classification where algorithms are used to perform feature selection, SVM does not need to do this in text classification.

- Text classification problem is linearly separable. SVM is the best classification algorithm if a problem is linear separable.
- Apart from C, we don't need to optimize any algorithm parameter, unlike other algorithms. We usually perform grid search to find the best set of parameters to use. This can be avoided in case of SVM.
- Training time for SVM is faster than other classification algorithms.

7.2 Data Preprocessing

The dataset obtained from TREC had certain junk characters in the form of Greek symbols such as Δ or ¶. So we removed the characters other than a-Z and A-Z. Removal of stop words was done during the training of the model itself. A stop words list was passed as a parameter during the training step.

After removal of the junk characters, the dataset looks like as shown in Figure 3,



```
DESC:manner How did serfdom develop in and then leave Russia ?
ENTY:cremat What films featured the character Popeye Doyle ?
DESC:manner How can I find a list of celebrities ' real names ?
ENTY:animal What fowl grabs the spotlight after the Chinese Year of the Monkey ?
ABBR:exp What is the full form of .com ?
HUM:ind What contemptible scoundrel stole the cork from my lunch ?
HUM:gr What team did baseball 's St. Louis Browns become ?
HUM:title What is the oldest profession ?
DESC:def What are liver enzymes ?
HUM:ind Name the scar-faced bounty hunter of The Old West .
NUM:date When was Ozzy Osbourne born ?
DESC:reason Why do heavier objects travel downhill faster ?
HUM:ind Who was The Pride of the Yankees ?
HUM:ind Who killed Gandhi ?
ENTY:event What is considered the costliest disaster the insurance industry has ever faced ?
LOC:state What sprawling U.S. state boasts the most airports ?
DESC:desc What did the only repealed amendment to the U.S. Constitution deal with ?
NUM:count How many Jews were executed in concentration camps during WWII ?
DESC:def What is '' Nine Inch Nails '' ?
DESC:def What is an annotated bibliography ?
NUM:date What is the date of Boxing Day ?
ENTY:other What articles of clothing are tokens in Monopoly ?
HUM:ind Name 11 famous martyrs .
DESC:desc What 's the Olympic motto ?
DESC:desc What is the origin of the name ' Scarlett ' ?
ENTY:letter What 's the second-most-used vowel in English ?
HUM:ind Who was the inventor of silly putty ?
LOC:other What is the highest waterfall in the United States ?
ENTY:other Name a golf course in Myrtle Beach .
LOC:state Which two states enclose Chesapeake Bay ?
ABBR:exp What does the abbreviation AIDS stand for ?
ENTY:other What does a spermologer collect ?
NUM:count How many points make up a perfect fivepin bowling score ?
HUM:gr Which company that manufactures video-game hardware sells the '' super system '' ?
NUM:count How many Community Chest cards are there in Monopoly ?
DESC:desc What do Mormons believe ?
NUM:date When did the neanderthal man live ?
```

Figure 3. Sample training dataset train_5500.txt

7.3 Feature Extraction

Feature extraction is the most important step in any predictive modeling. The processed dataset is imported and we use the advance natural processing techniques to extract the features for the model. We are using Python's Natural Language Toolkit (NLTK) library to extract most of the features. For text classification, following are features that we are interested in,

- **tf-idf Vectorizer-** tf-idf vectorizer helps in identifying the importance of a particular word in a text in the entire corpus. The importance of the word is decided by the frequency of appearance of the word in the corpus. This is done while calculating the term frequency. But we don't want to put more importance on the words such as "a", "the", "and", etc. After calculating the frequency of the words, in order to precisely assign the weight, we calculate the inverse document frequency to identify the most relevant words. The word that appears less is assigned more importance or weight. This feature makes the classifier understand what words to focus on while learning.

  While performing tf-idf, n-grams are also taken into consideration. N-grams helps in identifying which words are more frequently used to denote a person, or organization or location or any other entity. So it helps in identifying the importance of certain stop words while denoting an entity. The tf-idf score changes accordingly and it helps in better modeling of the predictive algorithm. In this project, we are using a standard *tf-idf()* function which is available in Python NLTK library.

- **Parts Of Speech (POS) tagging -** A POS tagger is software in which each word is assigned a parts of speech depending upon the neighboring word. As we also consider the adjacent word while tagging a particular word, n-grams are also considered while tagging a word. POS tagging helps in predicting what can be the next word in a series of concurrent words. It also helps as a basis of syntactic parsing and then meaning extraction. In the project, we are using a standard POS tagging function *pos_tag()* which is available in Python NLTK library. The parts of speech can be any of the following or a combination of it as shown in the diagram,

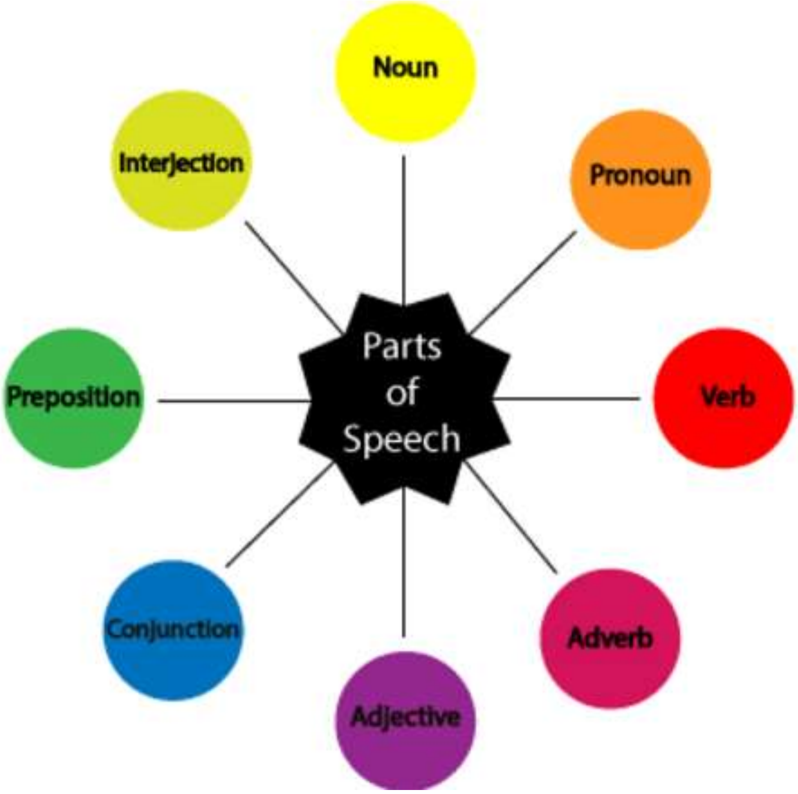
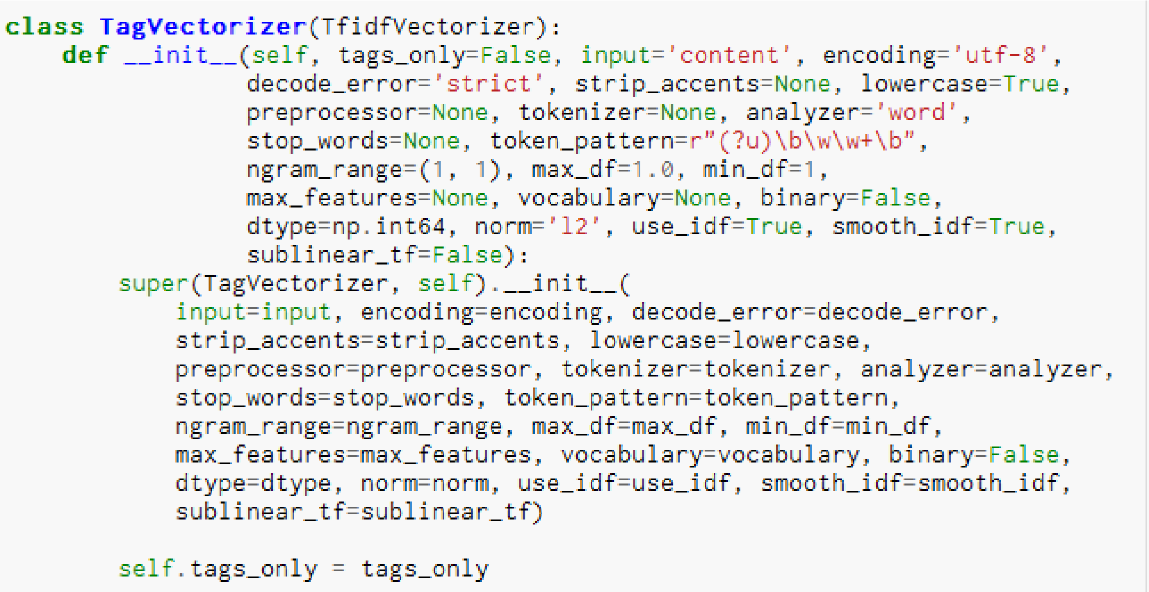
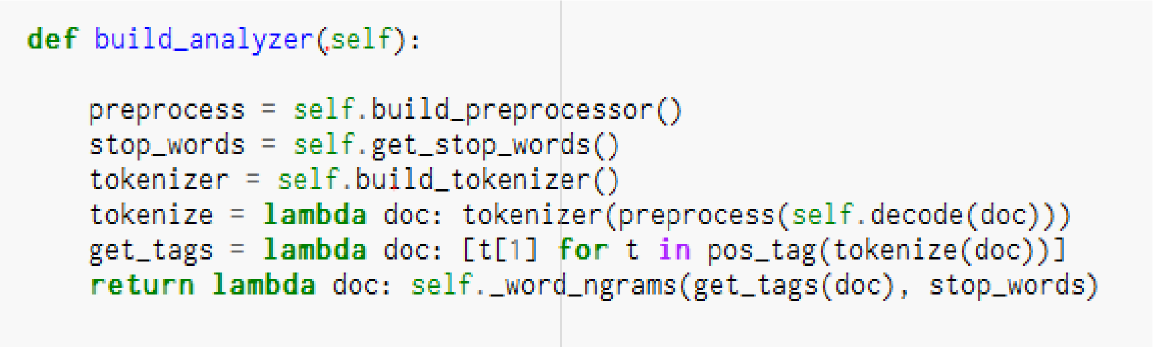


Figure 4. Different type of Parts of Speech



```
class TagVectorizer(TfidfVectorizer):
    def __init__(self, tags_only=False, input='content', encoding='utf-8',
                 decode_error='strict', strip_accents=None, lowercase=True,
                 preprocessor=None, tokenizer=None, analyzer='word',
                 stop_words=None, token_pattern=r"(?u)\b\w\w+\b",
                 ngram_range=(1, 1), max_df=1.0, min_df=1,
                 max_features=None, vocabulary=None, binary=False,
                 dtype=np.int64, norm='l2', use_idf=True, smooth_idf=True,
                 sublinear_tf=False):
        super(TagVectorizer, self).__init__(
            input=input, encoding=encoding, decode_error=decode_error,
            strip_accents=strip_accents, lowercase=lowercase,
            preprocessor=preprocessor, tokenizer=tokenizer, analyzer=analyzer,
            stop_words=stop_words, token_pattern=token_pattern,
            ngram_range=ngram_range, max_df=max_df, min_df=min_df,
            max_features=max_features, vocabulary=vocabulary, binary=False,
            dtype=dtype, norm=norm, use_idf=use_idf, smooth_idf=smooth_idf,
            sublinear_tf=sublinear_tf)

        self.tags_only = tags_only
```

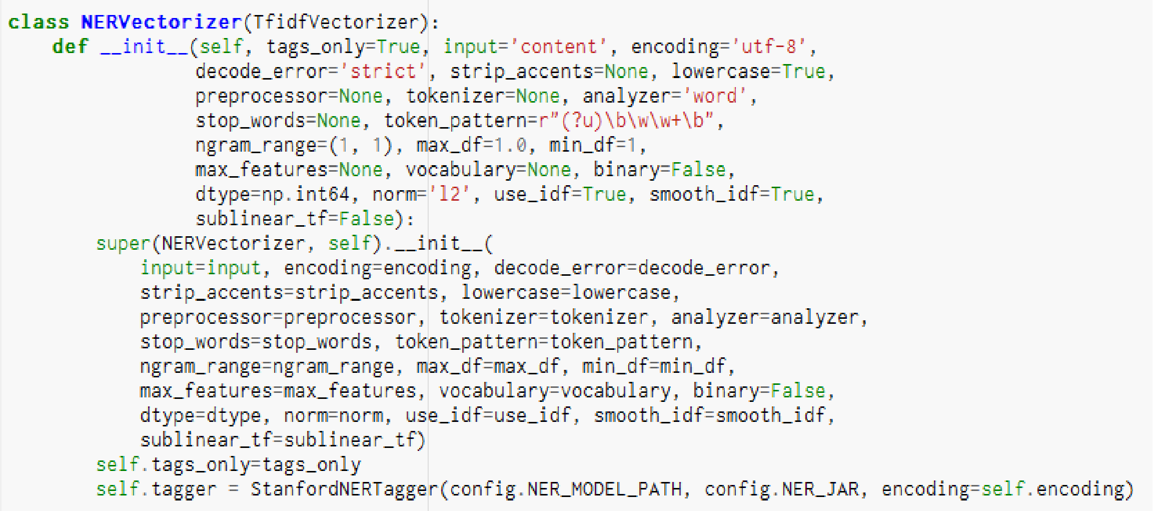
Figure 5. POS Tagger class



```
def build_analyzer(self):

    preprocess = self.build_preprocessor()
    stop_words = self.get_stop_words()
    tokenizer = self.build_tokenizer()
    tokenize = lambda doc: tokenizer(preprocess(self.decode(doc)))
    get_tags = lambda doc: [t[1] for t in pos_tag(tokenize(doc))]
    return lambda doc: self._word_ngrams(get_tags(doc), stop_words)
```

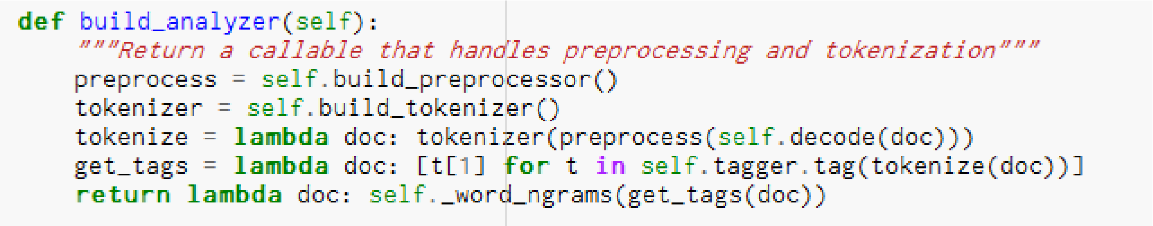
Figure 6. Extracting POS tags

- **Named Entity Recognition (NER)-** Named entity corresponds to classifying a word in any of the categories such as Persons, Products, Organizations, Locations etc. A NER tagger helps in classifying a word into any of the named entities. Extracting NER tags from a text helps in identifying whether a question is related to person, organization, or location or any of the six broad question type categories. For this project, we are using Stanford NER tagger which internally has functionality to extract the required named entity



```
class NERVectorizer(TfidfVectorizer):
    def __init__(self, tags_only=True, input='content', encoding='utf-8',
                 decode_error='strict', strip_accents=None, lowercase=True,
                 preprocessor=None, tokenizer=None, analyzer='word',
                 stop_words=None, token_pattern=r"(?u)\b\w\w+\b",
                 ngram_range=(1, 1), max_df=1.0, min_df=1,
                 max_features=None, vocabulary=None, binary=False,
                 dtype=np.int64, norm='l2', use_idf=True, smooth_idf=True,
                 sublinear_tf=False):
        super(NERVectorizer, self).__init__(
            input=input, encoding=encoding, decode_error=decode_error,
            strip_accents=strip_accents, lowercase=lowercase,
            preprocessor=preprocessor, tokenizer=tokenizer, analyzer=analyzer,
            stop_words=stop_words, token_pattern=token_pattern,
            ngram_range=ngram_range, max_df=max_df, min_df=min_df,
            max_features=max_features, vocabulary=vocabulary, binary=False,
            dtype=dtype, norm=norm, use_idf=use_idf, smooth_idf=smooth_idf,
            sublinear_tf=sublinear_tf)
        self.tags_only=tags_only
        self.tagger = StanfordNERTagger(config.NER_MODEL_PATH, config.NER_JAR, encoding=self.encoding)
```

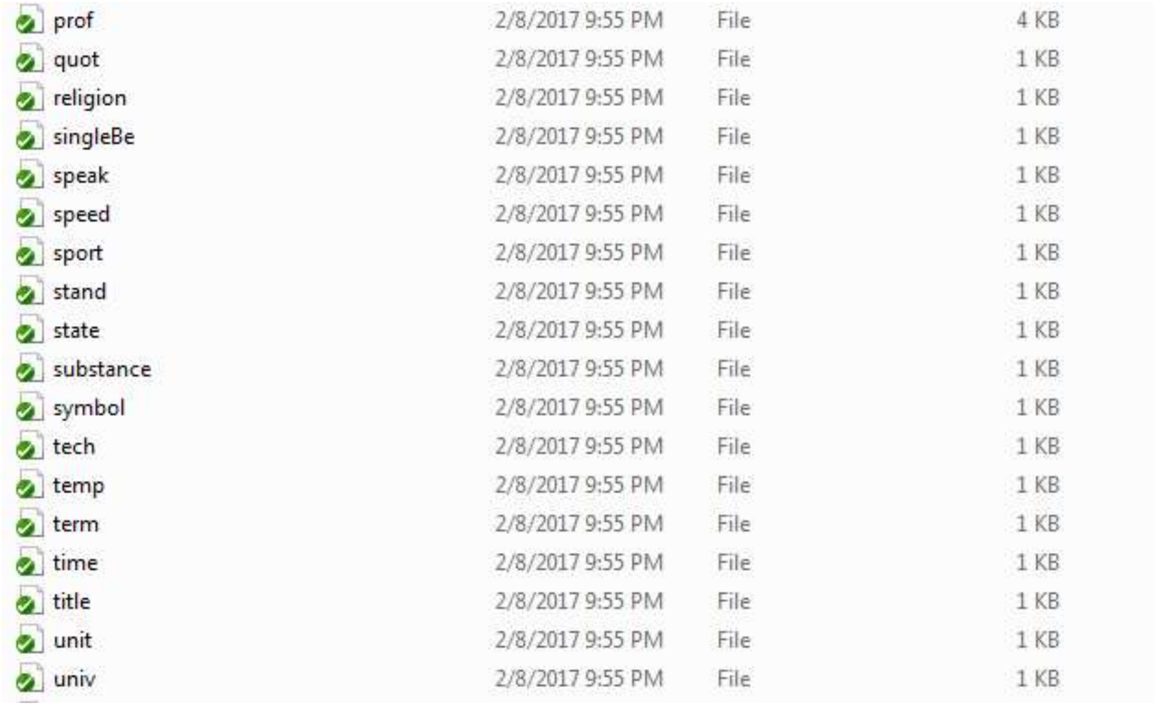
Figure 7. NER Tagger class



```
def build_analyzer(self):
    """Return a callable that handles preprocessing and tokenization"""
    preprocess = self.build_preprocessor()
    tokenizer = self.build_tokenizer()
    tokenize = lambda doc: tokenizer(preprocess(self.decode(doc)))
    get_tags = lambda doc: [t[1] for t in self.tagger.tag(tokenize(doc))]
    return lambda doc: self._word_ngrams(get_tags(doc))
```

Figure 8. Tokenizing and extracting NER tags

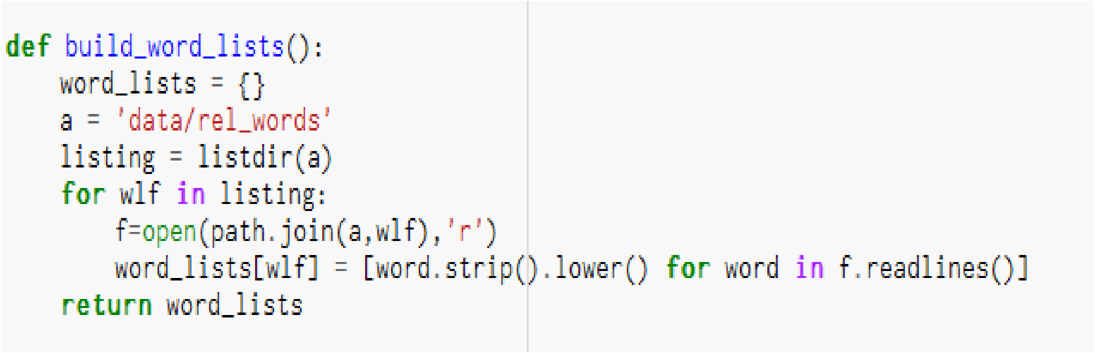
- **Semantically Related Words-** Performing semantic analysis is one of the important steps in text classification. A single sentence can be modified and represented in multiple ways. It is important to analyze the meaning of the sentence no matter how it is represented. In this project, we are providing a word list which consists a directory of most commonly used words and its different representations. Sample directory structure is shown below,



prof	2/8/2017 9:55 PM	File	4 KB
quot	2/8/2017 9:55 PM	File	1 KB
religion	2/8/2017 9:55 PM	File	1 KB
singleBe	2/8/2017 9:55 PM	File	1 KB
speak	2/8/2017 9:55 PM	File	1 KB
speed	2/8/2017 9:55 PM	File	1 KB
sport	2/8/2017 9:55 PM	File	1 KB
stand	2/8/2017 9:55 PM	File	1 KB
state	2/8/2017 9:55 PM	File	1 KB
substance	2/8/2017 9:55 PM	File	1 KB
symbol	2/8/2017 9:55 PM	File	1 KB
tech	2/8/2017 9:55 PM	File	1 KB
temp	2/8/2017 9:55 PM	File	1 KB
term	2/8/2017 9:55 PM	File	1 KB
time	2/8/2017 9:55 PM	File	1 KB
title	2/8/2017 9:55 PM	File	1 KB
unit	2/8/2017 9:55 PM	File	1 KB
univ	2/8/2017 9:55 PM	File	1 KB

Figure 9. Directory structure of related word list

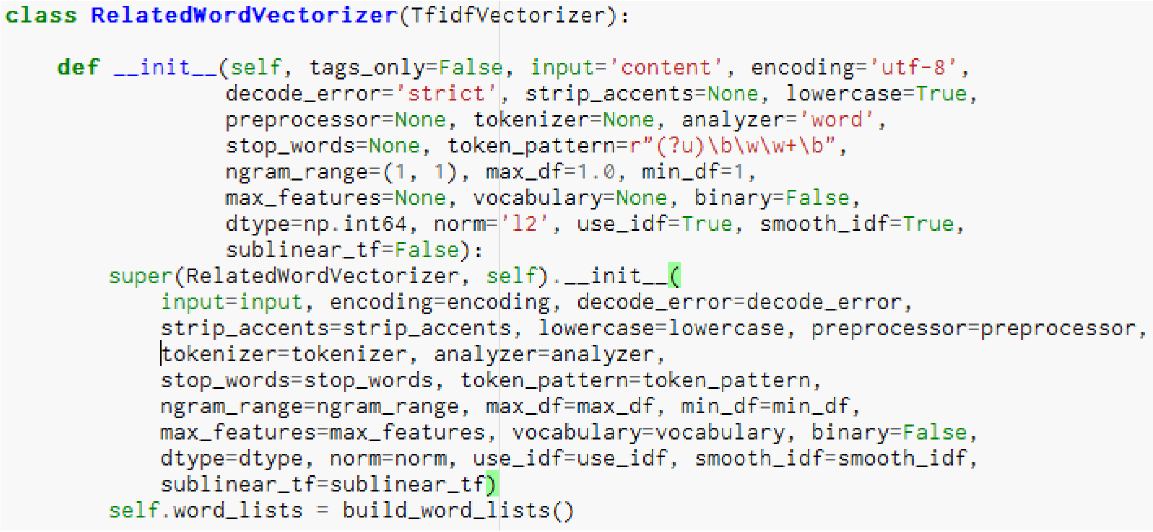
First we create a dictionary of all the related words with key as the file_name and value as a list all the words inside that file. This process can be shown using the following implementation,



```
def build_word_lists():
    word_lists = {}
    a = 'data/rel_words'
    listing = listdir(a)
    for wlf in listing:
        f=open(path.join(a,wlf),'r')
        word_lists[wlf] = [word.strip().lower() for word in f.readlines()]
    return word_lists
```

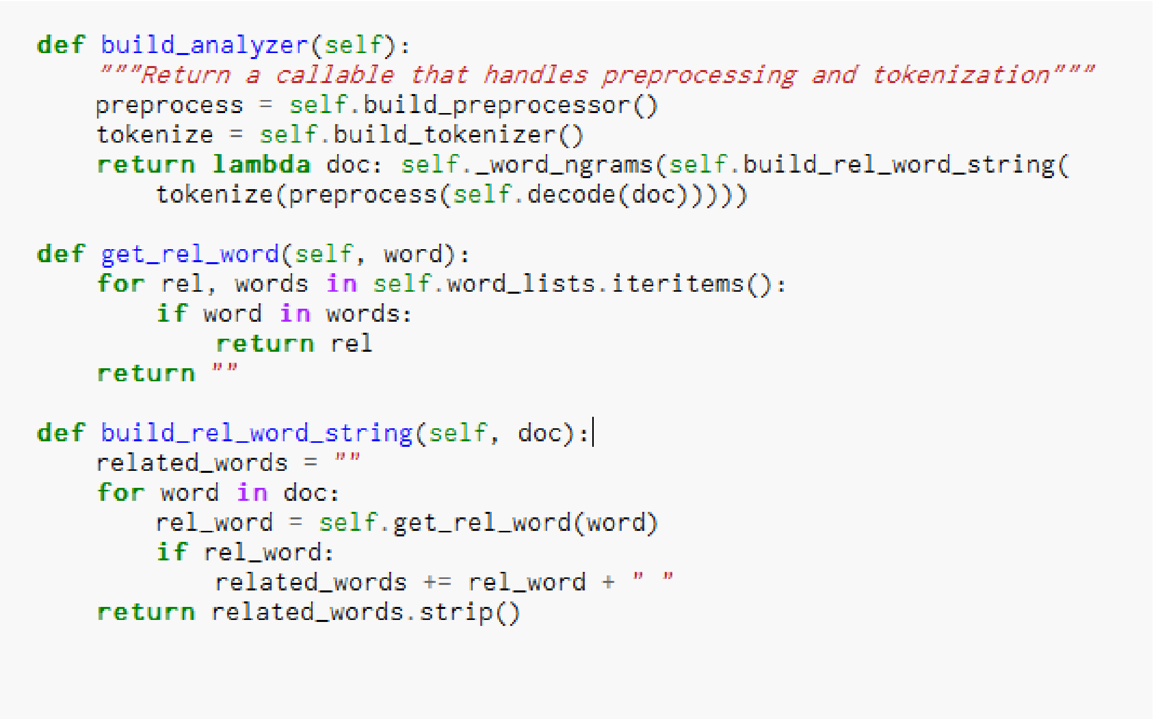
Figure 9. Building a word list from the related words directory

Then for each tokenized word of the text, we compare the word with this dictionary to check in which category (which is the key of the dictionary) does the tokenized word fall. Once we find the all the keywords, we create a string that resembles all the category and pass it as a feature. So no matter how differently a sentence is constructed, the semantics will always be the same. This reduces the learning efforts of the classifier as it does not have to learn and recognize each sentence in a different manner. By doing semantic analysis, a classifier can recognize any semantically related sentence to what it learned during training phase. This process is shown in the following implementation,



```
class RelatedWordVectorizer(TfidfVectorizer):

    def __init__(self, tags_only=False, input='content', encoding='utf-8',
                 decode_error='strict', strip_accents=None, lowercase=True,
                 preprocessor=None, tokenizer=None, analyzer='word',
                 stop_words=None, token_pattern=r"(?u)\b\w\w+\b",
                 ngram_range=(1, 1), max_df=1.0, min_df=1,
                 max_features=None, vocabulary=None, binary=False,
                 dtype=np.int64, norm='l2', use_idf=True, smooth_idf=True,
                 sublinear_tf=False):
        super(RelatedWordVectorizer, self).__init__(
            input=input, encoding=encoding, decode_error=decode_error,
            strip_accents=strip_accents, lowercase=lowercase, preprocessor=preprocessor,
            tokenizer=tokenizer, analyzer=analyzer,
            stop_words=stop_words, token_pattern=token_pattern,
            ngram_range=ngram_range, max_df=max_df, min_df=min_df,
            max_features=max_features, vocabulary=vocabulary, binary=False,
            dtype=dtype, norm=norm, use_idf=use_idf, smooth_idf=smooth_idf,
            sublinear_tf=sublinear_tf)
        self.word_lists = build_word_lists()
```

Figure 10. Related Word Vectorizer Class



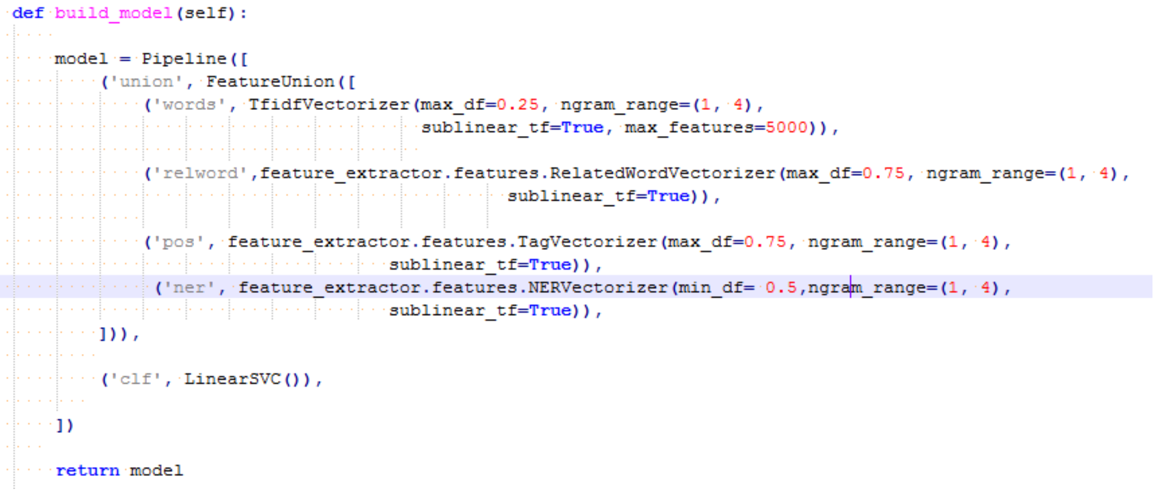
```
def build_analyzer(self):
    """Return a callable that handles preprocessing and tokenization"""
    preprocess = self.build_preprocessor()
    tokenize = self.build_tokenizer()
    return lambda doc: self._word_ngrams(self.build_rel_word_string(
        tokenize(preprocess(self.decode(doc)))))

def get_rel_word(self, word):
    for rel, words in self.word_lists.iteritems():
        if word in words:
            return rel
    return ""

def build_rel_word_string(self, doc):
    related_words = ""
    for word in doc:
        rel_word = self.get_rel_word(word)
        if rel_word:
            related_words += rel_word + " "
    return related_words.strip()
```

Figure 11. Building string for related words

7.4 Predictive Modeling

After extracting all the features, we create a model using a machine learning classification algorithm Support Vector Machine (SVM). The model is created using a function called as *pipeline()*. In the *pipeline()* function, we give a list of transformers and a estimator. Estimator can be any machine learning algorithm, in this case it is SVM. Pipeline helps in reducing the necessity to call the *transform()* method after every transformation. All the transformations are done when we call the *fit()* method. Finally the estimator, which has the *fit()* method gets the final set of features that is needed to train the model.

Following is the code snippet for the pipeline implementation,



```
def build_model(self):

    model = Pipeline([
        ('union', FeatureUnion([
            ('words', TfidfVectorizer(max_df=0.25, ngram_range=(1, 4),
                                      sublinear_tf=True, max_features=5000)),

            ('relword',feature_extractor.features.RelatedWordVectorizer(max_df=0.75, ngram_range=(1, 4),
                                              sublinear_tf=True)),

            ('pos', feature_extractor.features.TagVectorizer(max_df=0.75, ngram_range=(1, 4),
                                   sublinear_tf=True)),
             ('ner', feature_extractor.features.NERVectorizer(min_df= 0.5,ngram_range=(1, 4),
                                   sublinear_tf=True)),
        ])),

        ('clf', LinearSVC()),

    ])

    return model
```

Figure 12. Pipeline implementation

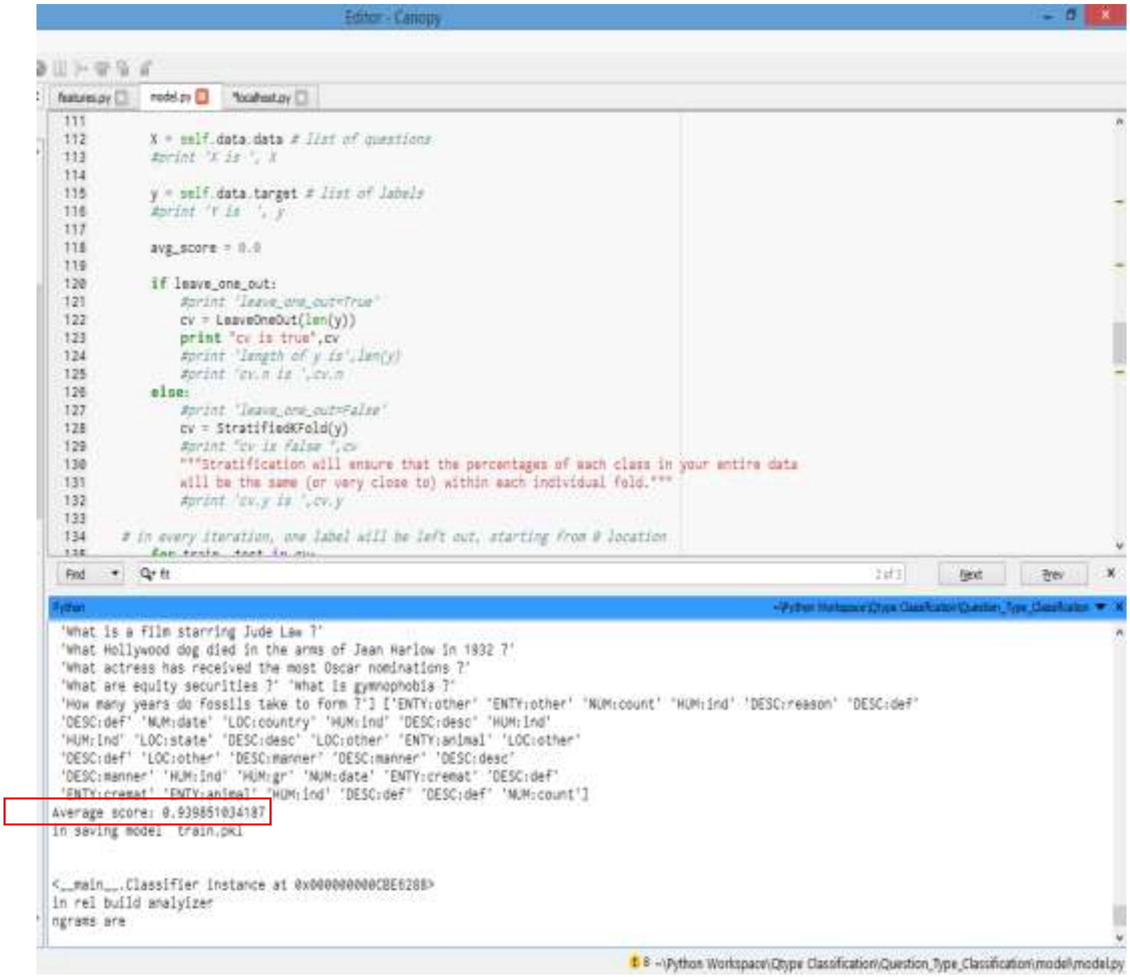
FeatureUnion() estimator function helps in concatenating the output of all the transformations. The transformations are applied in parallel for each input data and then it concatenates the results in the end. This final feature set is given to the estimator, which can be any machine learning algorithm.

After all the pre-processing and the feature extraction phases, the data point includes

- Named entity for each tokenized words,
- The tf-idf vectors,
- POS tags for each word and,
- The set of related words from the bag of words for each tokenized word

Additionally, the n-gram parameter has been set to take values between 1 to 4 while creating the model pipeline. This enhances the learning process for the model.

After we train the model , the model is evaluated using the TREC_10 dataset. The model gave an accuracy of 93%.



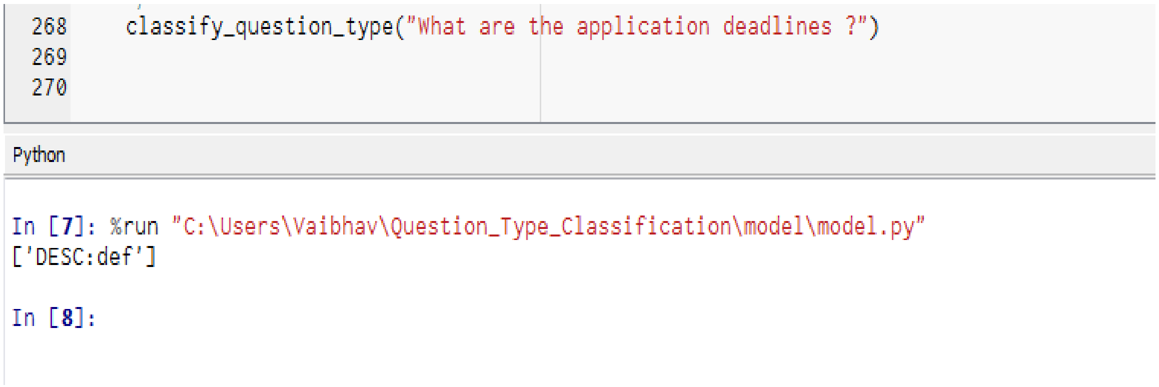
Editor - Canopy
features.py
model.py
*localhost.py
Find
ft
2 of 3
Next
Prev
Python
~\Python Workspace\Qtype Classification\Question_Type_Classification
~\Python Workspace\Qtype Classification\Question_Type_Classification\model\model.py

```
        X = self.data.data # list of questions
        #print 'X is ', X

        y = self.data.target # list of labels
        #print 'Y is  ', y

        avg_score = 0.0

        if leave_one_out:
            #print 'leave_one_out=True'
            cv = LeaveOneOut(len(y))
            print "cv is true",cv
            #print 'length of y is',len(y)
            #print 'cv.n is ',cv.n
        else:
            #print 'leave_one_out=False'
            cv = StratifiedKFold(y)
            #print "cv is false ",cv
            """Stratification will ensure that the percentages of each class in your entire data
            will be the same (or very close to) within each individual fold."""
            #print 'cv.y is ',cv.y

    # in every iteration, one label will be left out, starting from 0 location
        for train, test in cv:
```

```
 'What is a film starring Jude Law ?'
 'What Hollywood dog died in the arms of Jean Harlow in 1932 ?'
 'What actress has received the most Oscar nominations ?'
 'What are equity securities ?' 'What is gymnophobia ?'
 'How many years do fossils take to form ?'] ['ENTY:other' 'ENTY:other' 'NUM:count' 'HUM:ind' 'DESC:reason' 'DESC:def'
 'DESC:def' 'NUM:date' 'LOC:country' 'HUM:ind' 'DESC:desc' 'HUM:ind'
 'HUM:ind' 'LOC:state' 'DESC:desc' 'LOC:other' 'ENTY:animal' 'LOC:other'
 'DESC:def' 'LOC:other' 'DESC:manner' 'DESC:manner' 'DESC:desc'
 'DESC:manner' 'HUM:ind' 'HUM:gr' 'NUM:date' 'ENTY:cremat' 'DESC:def'
 'ENTY:cremat' 'ENTY:animal' 'HUM:ind' 'DESC:def' 'DESC:def' 'NUM:count']
Average score: 0.939851034187
in saving model  train.pkl


<__main__.Classifier instance at 0x000000000CBE6288>
in rel build analyizer
ngrams are
```

Figure 13. Evaluation results

The model is now ready for making prediction. Following code snippet shows a sample run of the prediction,

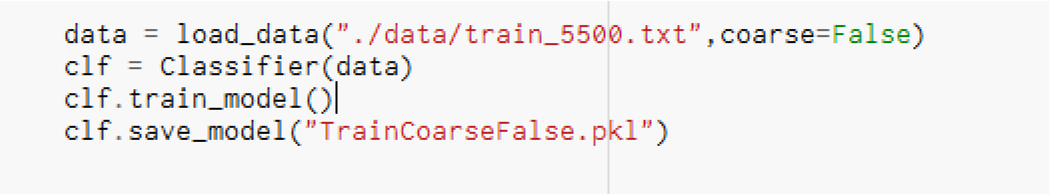


```
classify_question_type("What are the application deadlines ?")
```

```
In [7]: %run "C:\Users\Vaibhav\Question_Type_Classification\model\model.py"
['DESC:def']

In [8]:
```

Figure 14. Sample run for making prediction

After creating the model we save the model so that we can use it for later purpose.



```
data = load_data("./data/train_5500.txt",coarse=False)
clf = Classifier(data)
clf.train_model()
clf.save_model("TrainCoarseFalse.pkl")
```

Figure 15. Saving the model

As a part of the real life application, the trained classifier was integrated with Slack. So instead of making prediction by taking user inputs from the console, we created a bot application on Slack.

7.5 Application using Slack

Slack is an online messaging platform. It helps the teams to collaborate and communicate at the same time. The idea behind creating a Slack bot was to create an application where we can use our classifier. So we decided to create a bot using the botkit framework [6]. Botkit is a library that is used to create conversation bots by easing the process of creation. Botkit has compatibility with different platforms such as Slack, Twilio, Microsoft bot framework and Facebook.

7.5.1 Implementation of the bot

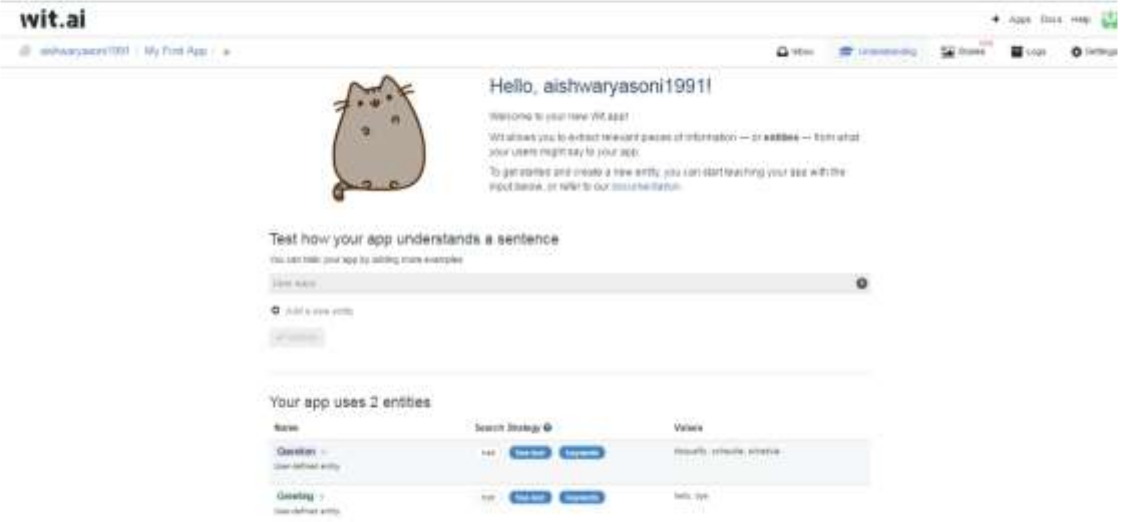
The bot was developed in the two phases,

1. Using wit.ai
2. Using the classifier by writing a custom middleware

7.5.1.1 Implementation using Wit.ai

Initially, before the classifier was developed, a simple bot was created using wit.ai. Wit provides an online console and API's to create a smarter conversation bot. It uses NLP techniques to make an intelligent bot. It maps the user inputs with the predefined entities and intents and gives the output accordingly. Initially a user create a list of entities and intents to train the wit model. For training, the user enters inputs from Slack console, and the question comes to the message folder of the wit.ai. The user validates the message by mapping it to an appropriate intent. Then after sufficient amount of training, the model starts predicting the intents for unknown inputs. For example, for a "**food**" intent, users may say "Can someone order the **food**?" If we don't find an existing intent, or if there is no suggested intent we can create our own intent.

For demo purpose, we created two entities, "**Greeting**" and "**Question**". We took two questions from the CS FAQ section to determine the intent and retrieve the answers to it. The questions focused on determining the rules for disqualifications and how to schedule appointments with the advisor. The following images show sample screenshots of the entities the keywords and the expressions used to train the model. The keywords help to identify the intent and map it to the appropriate entity.



Hello, aishwaryasoni1991!

Welcome to your new Wit app!

Wit allows you to extract relevant pieces of information — or **entities** — from what your users might say to your app.

To get started and create a new entity, you can start teaching your app with the input below, or refer to our documentation.

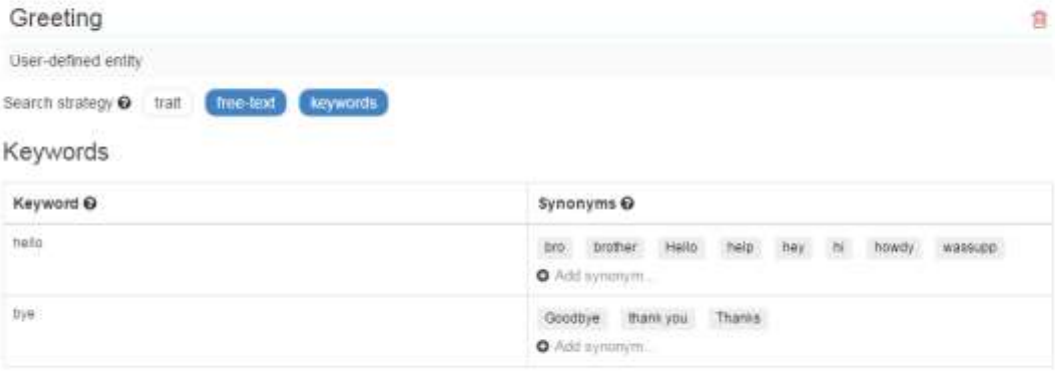
Test how your app understands a sentence

You can train your app by adding more examples

Your app uses 2 entities

Name	Search Strategy	Values
Question User defined entity	trait Free text Keywords	disqualify, schedule, advisor
Greeting User defined entity	trait Free text Keywords	hello, bye

Figure 16. Creating sample entities on wit console



Greeting

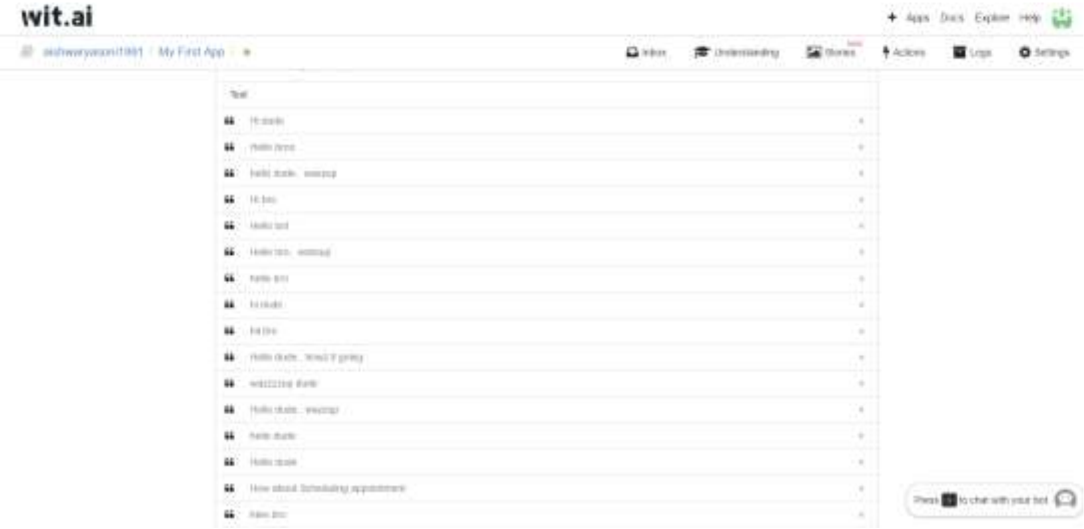
User-defined entity

Search strategy ❓ trait **free-text** **keywords**

Keywords

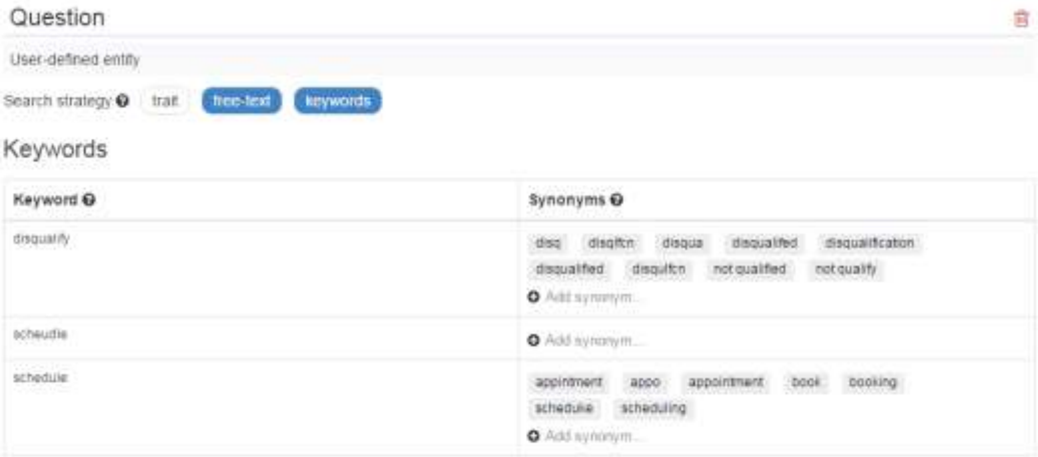
Keyword ❓	Synonyms ❓
hello	bro brother Hello help hey hi howdy wassupp Add synonym...
bye	Goodbye thank you Thanks Add synonym...

Figure 17. Keywords for "Greeting" entity



Text
Hi dude
Hello bros
hello dude.. wassup
Hi bro
Hello bot
Hello bro.. wassup
hello bro
hi dude
hi bro
Hello dude.. how's it going
wassup dude
Hello dude.. wassup
hello dude
Hello dude
How about Scheduling appointment
[illegible] bro

Figure 18. Expressions for "Greeting" entity



Question

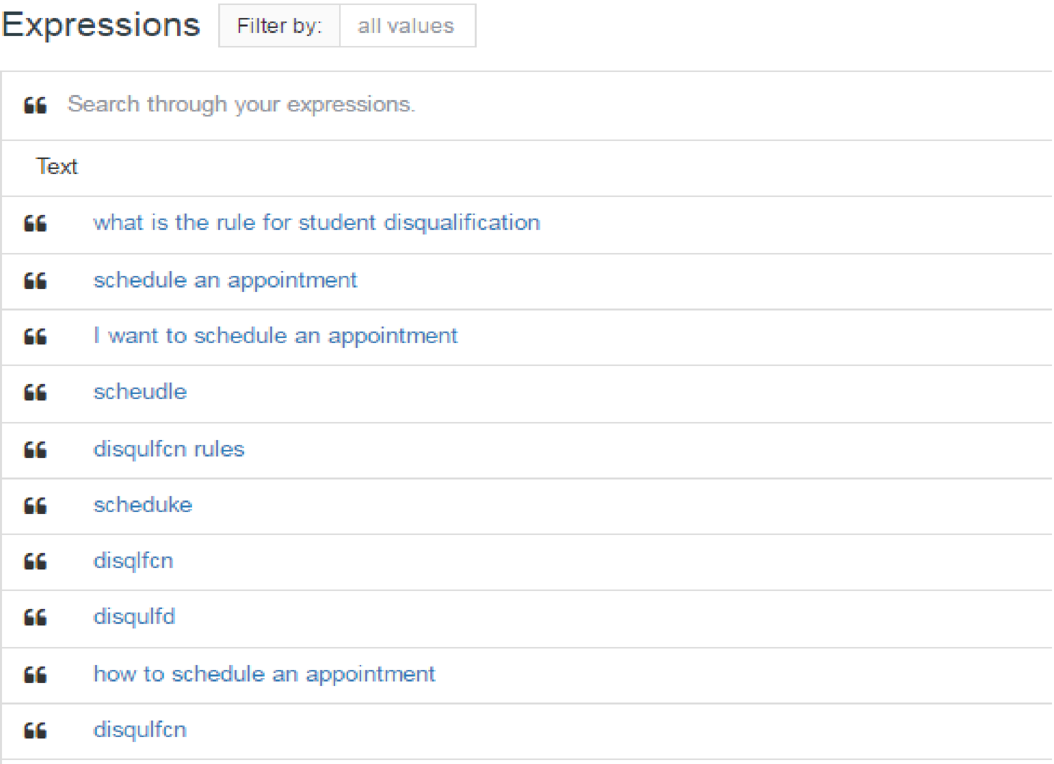
User-defined entity

Search strategy ❓ trait **free-text** **keywords**

Keywords

Keyword ❓	Synonyms ❓
disqualify	disq disqlfcn disqua disqualifed disqualification disqualified disqulfcn not qualifed not qualify Add synonym...
scheudle	Add synonym...
schedule	appintment appo appointment book booking scheduke scheduling Add synonym...

Figure 19. Keywords for "Question" entity



Expressions

Filter by: all values

Search through your expressions.

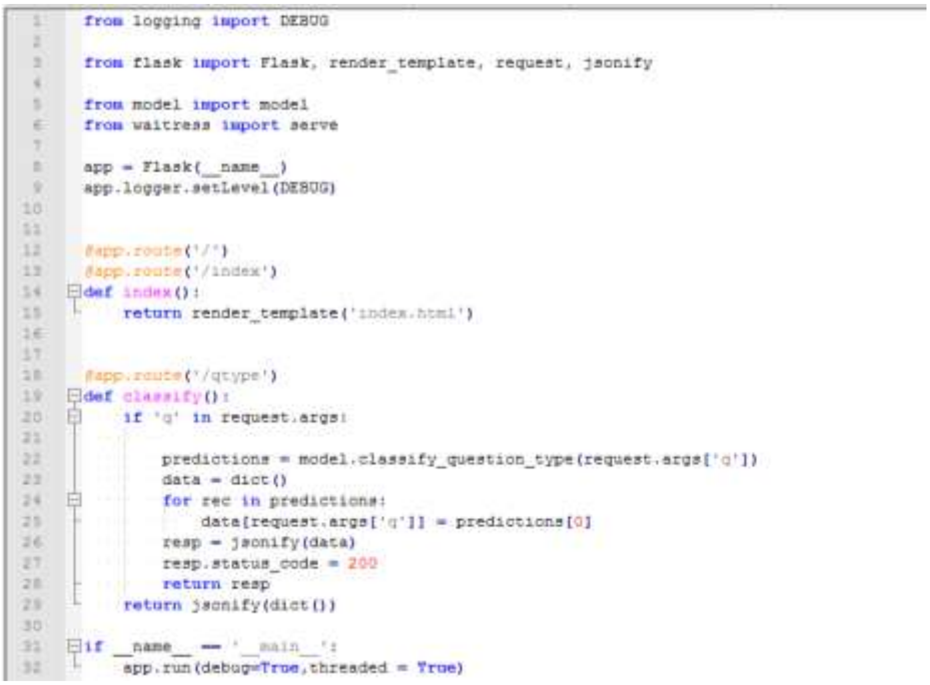
Text
what is the rule for student disqualification
schedule an appointment
I want to schedule an appointment
scheudle
disqulfcn rules
scheduke
disqlfcn
disqulfd
how to schedule an appointment
disqulfcn

Figure 20. Expressions for "Question" entity

The next step was to integrate wit with the botkit framework. We used a middleware service that connects both wit and the botkit. Now, every message that was received to our bot was sent to wit.ai for processing. After the processing, the output from wit comes in the form of JSON. The middleware parses through the JSON output and finds the key "**message.entities**" to get the intent behind the question.

7.5.1.2 Implementation using the Classifier

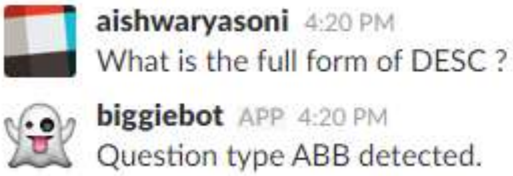
The aim was to build a real life application using the classifier. To integrate the classifier with Slack, we wrote a custom middleware. So instead of going to wit.ai, the request now goes to the classifier application. For this, it was important to create a REST based application of our implementation. We used Flask to create a REST service. It is a framework which is available in Python and it helps in creating web services. Figure given below shows the implementation of our application using Flask.



```
from logging import DEBUG

from flask import Flask, render_template, request, jsonify

from model import model
from waitress import serve

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


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


@app.route('/qtype')
def classify():
    if 'q' in request.args:

        predictions = model.classify_question_type(request.args['q'])
        data = dict()
        for rec in predictions:
            data[request.args['q']] = predictions[0]
        resp = jsonify(data)
        resp.status_code = 200
        return resp
    return jsonify(dict())

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

Figure 21. Implementation using Flask

It looks for '**q**' in the arguments of the URL which holds the question. The question is then forwarded to the implementation where we have the classifier and after the classification process, the output is send back to the Slack bot.

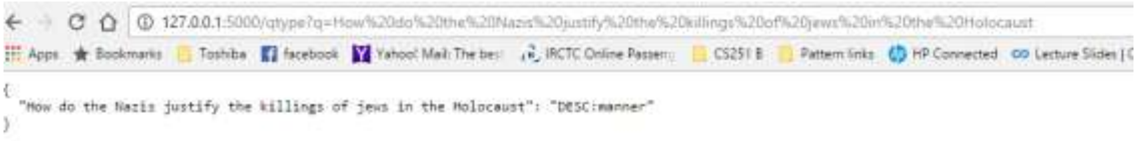


aishwaryasoni 4:20 PM
What is the full form of DESC ?

biggiebot APP 4:20 PM
Question type ABB detected.

Figure 22. Sample run of the Slack bot

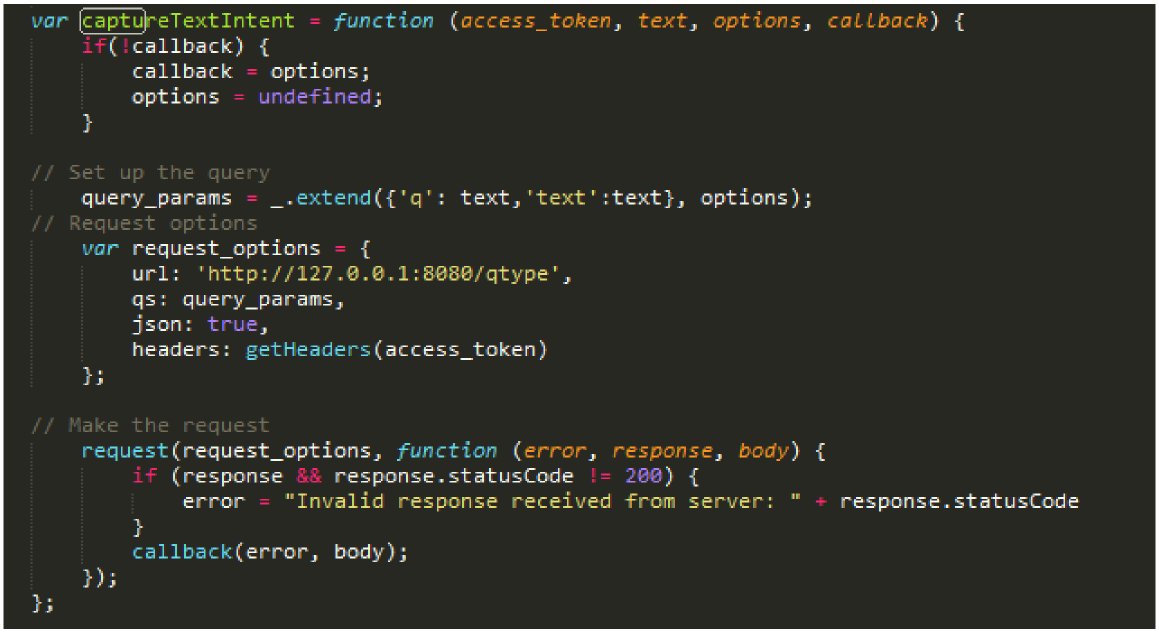
We can also check the console of our REST application to see the output of our question. For example, the figure below shows how the REST service behaves when it receives any question from the Slack bot,



```
{
  "How do the Nazis justify the killings of jews in the Holocaust": "DESC:manner"
}
```

Figure 23. Output of the REST application

To interact with the REST service, we wrote a custom middleware service. This middleware helps in managing the request from Slack and response from our REST application. In the middleware implementation, we specified the URL where the request from Slack should go to. This URL is the IP address and the port number where our REST application runs. We also specify what query parameter to look for in the request, here it is "**q**". The following images show the code snippets for the custom middleware implementation,

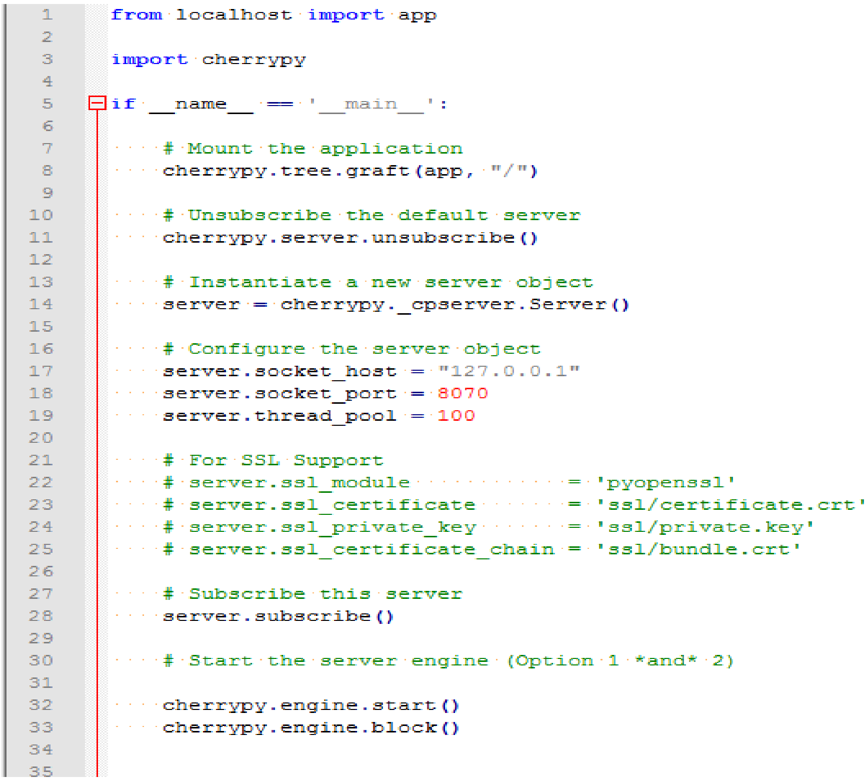


```
var captureTextIntent = function (access_token, text, options, callback) {
    if(!callback) {
        callback = options;
        options = undefined;
    }

// Set up the query
    query_params = _.extend({'q': text,'text':text}, options);
// Request options
    var request_options = {
        url: 'http://127.0.0.1:8080/qtype',
        qs: query_params,
        json: true,
        headers: getHeaders(access_token)
    };

// Make the request
    request(request_options, function (error, response, body) {
        if (response && response.statusCode != 200) {
            error = "Invalid response received from server: " + response.statusCode
        }
        callback(error, body);
    });
};
```

Figure 24. Code for the middleware implementation

7.6 Multiprocessing Using CherryPy WSGI server

The application created using Flask was a sequential application. In order to enable multiprocessing, we used a web framework for Python called as CherryPy. It is Web Server Gateway Interface (WSGI) framework which provides a multi-threaded web server where we can deploy any application. The main advantage of using CherryPy is that it is a very easy to use framework with only few customization needed to run any application. When the REST application is deployed to CherryPy, the application runs in a thread. It helps in utilizing all the cores of the CPU's and thereby handling multiple requests. The following code snippet shows the implementation using CherryPy WSGI server. As shown, the "app" in the import statement is the name of the Flask application that we need to deploy on CherryPy.

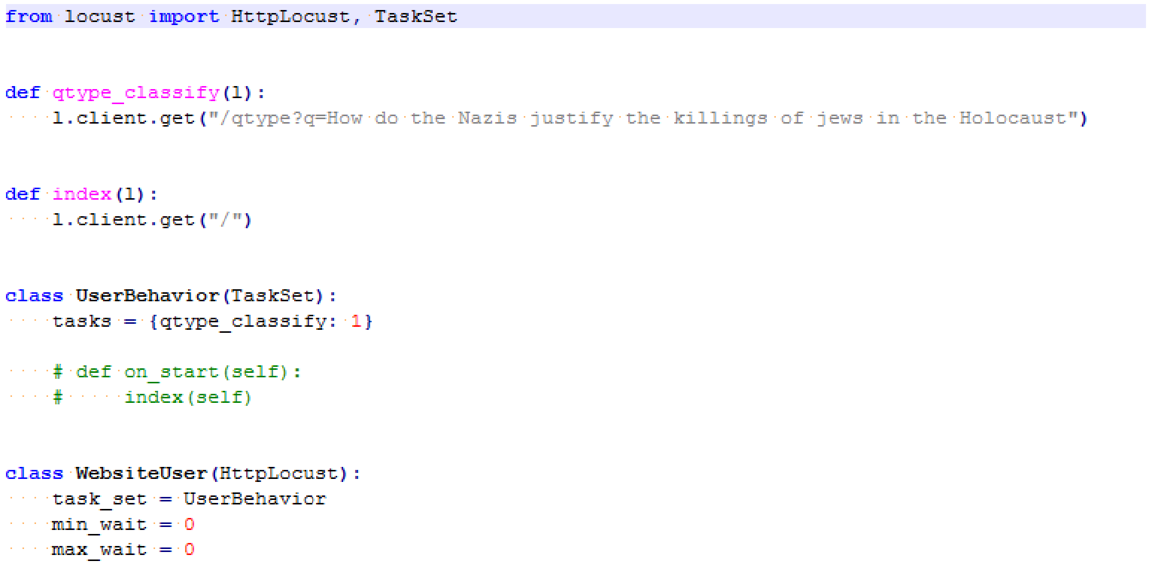


```
from localhost import app

import cherrypy

if __name__ == '__main__':

    # Mount the application
    cherrypy.tree.graft(app, "/")

    # Unsubscribe the default server
    cherrypy.server.unsubscribe()

    # Instantiate a new server object
    server = cherrypy._cpserver.Server()

    # Configure the server object
    server.socket_host = "127.0.0.1"
    server.socket_port = 8070
    server.thread_pool = 100

    # For SSL Support
    # server.ssl_module            = 'pyopenssl'
    # server.ssl_certificate       = 'ssl/certificate.crt'
    # server.ssl_private_key       = 'ssl/private.key'
    # server.ssl_certificate_chain = 'ssl/bundle.crt'

    # Subscribe this server
    server.subscribe()

    # Start the server engine (Option 1 *and* 2)

    cherrypy.engine.start()
    cherrypy.engine.block()

```

Figure 25. CherryPy implementation

7.7 Load Testing the application

To test how much our application can take the load, we used locust.io. It is a tool to load test any python application. Following is the implementation snippet for our application,



```
from locust import HttpLocust, TaskSet


def qtype_classify(l):
    l.client.get("/qtype?q=How do the Nazis justify the killings of jews in the Holocaust")


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


class UserBehavior(TaskSet):
    tasks = {qtype_classify: 1}

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


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

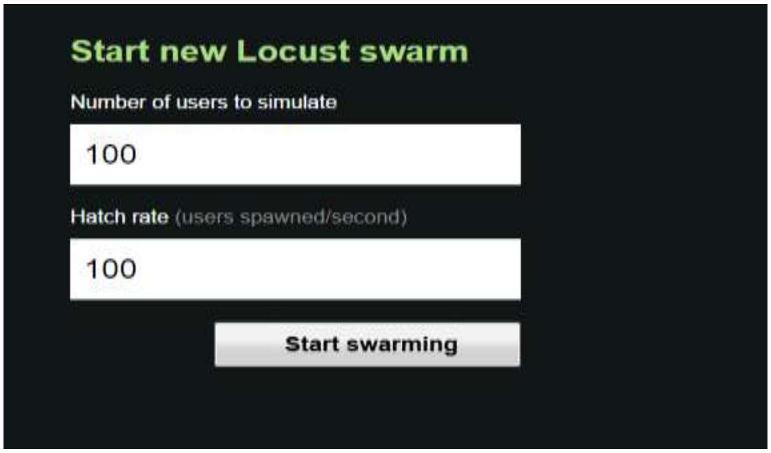
Figure 26. Locust file for load testing

The name of our locust file is server.py. To run the file we use the following command,

" locust -f testing/server.py --host=127.0.0.1:5000 "

The host is the URL where the locust server is running. In the code, we have set the waiting time for a request to come as 0.

To perform load testing, we simulate the requests from 100 users. The number of users per second (hatch rate) keep on increasing from 1 till it reaches 100. Figure shown below shows the initial console of locust.io where can enter our values for users and the hatch rate,



Start new Locust swarm

Number of users to simulate

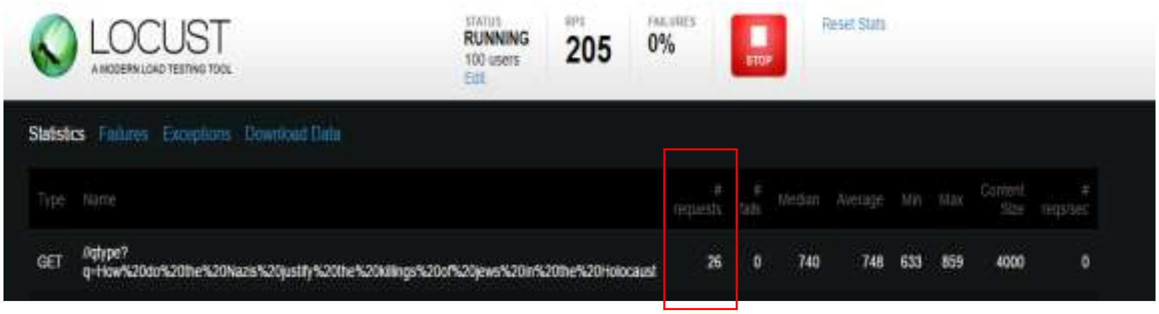
100

Hatch rate (users spawned/second)

100

Figure Sample run of load test

After swarming the requests, we check how the application behaves. As shown in the figure below, the application dies after taking 26 simultaneous requests. It doesn't process any request further than 26.



STATUS RUNNING 100 users

RPS 205

FAILURES 0%

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
GET	/qtype?q=How%20do%20the%20Nazis%20justify%20the%20killings%20of%20jews%20in%20the%20Holocaust	26	0	740	748	633	859	4000	0

Figure Load testing results

As we can see, the performance was still a concern. It was necessary to build an application that can handle huge load and achieves the goal of scalability. To solve the problem of scalability and performance we moved our focus to Apache Spark.

8. Implementation Using Apache Spark

Spark is a framework which is used to process huge amount of data in a fast and very efficient manner. It was developed at Berkley in 2009 and it soon became a top-level project at Apache. Following are some of the advantages of using Apache Spark,

- Fast and reliable in memory processing (100x faster than traditional MapReduce)
- Fault tolerant capability as Spark uses an immutable data structure called as RDD
- It provides MLlib as one of the libraries using which we can perform any machine learning techniques
- It provides the ability to perform real-time streaming using Spark Streaming

8.1 Overview of Spark Architecture

The following figure shows the architecture of Apache Spark [7],

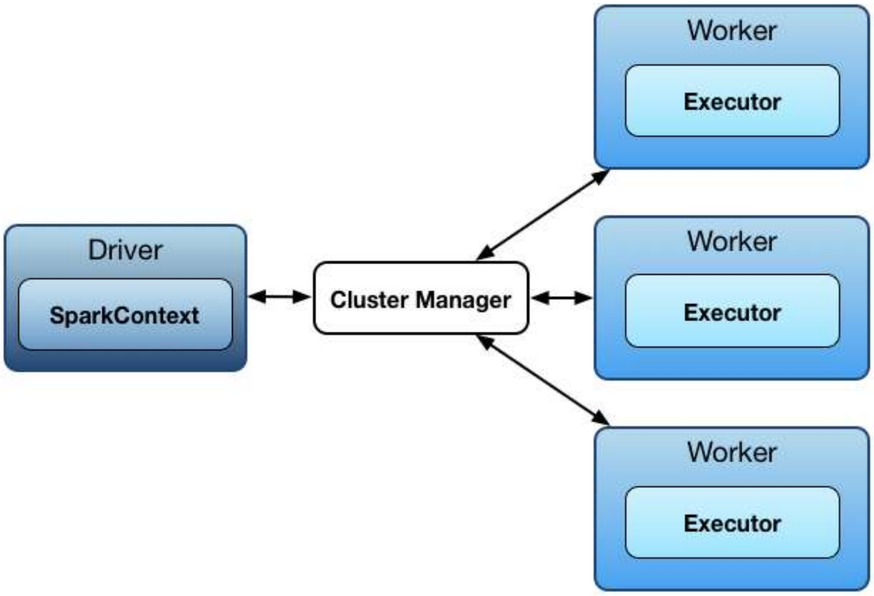


Figure Spark Architecture

Driver is the master node that controls the execution and distribution of tasks across all worker nodes. SparkContext is used to create an instance of Spark application. Using that instance we perform any functions, like creating RDDs, using broadcast variables, creating jobs and accessing spark services until the instance is stopped. Cluster manager can be Mesos, Yarn or Hadoop itself. We can also run Spark in a standalone mode. When the task is ready for execution, driver distributes the task to all the worker nodes. Each worker has an executor which executes the assigned task. After the completion of the task, the result is sent back to the driver daemon. The figure below shows the stack that spark supports on its architecture [8],

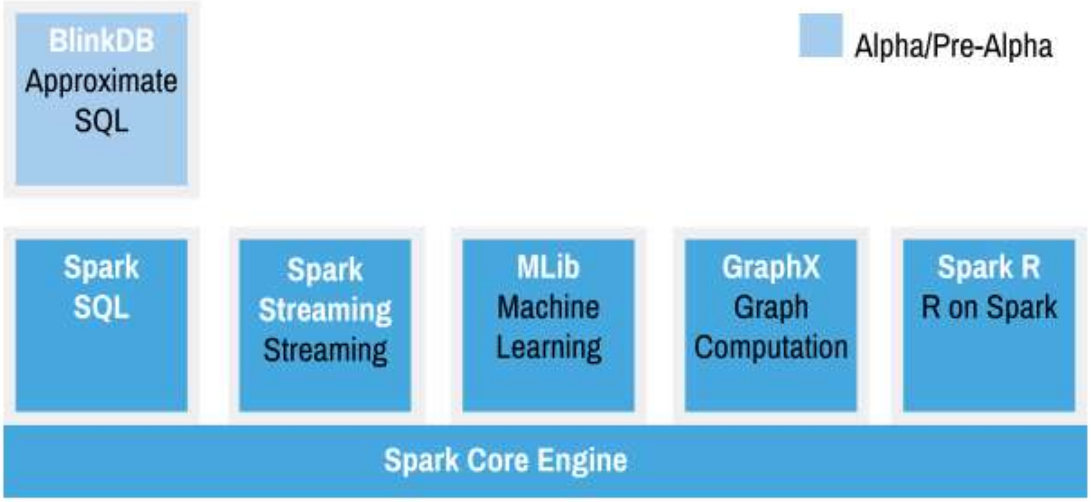
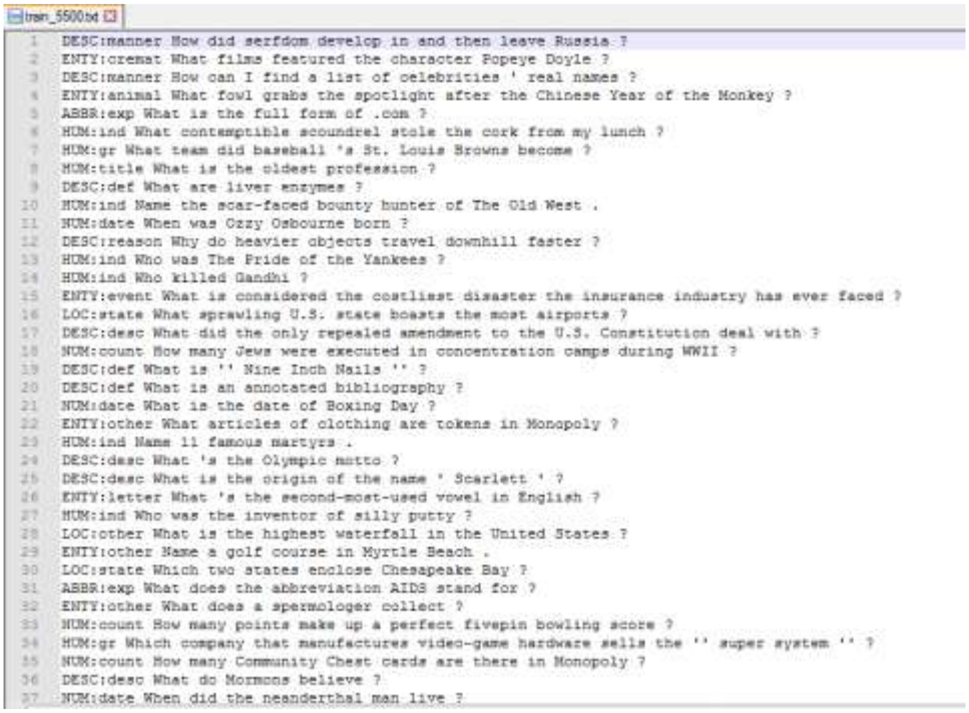


Figure Spark Stack

8.2 Data Preprocessing

We are using the same pre-processed data obtained from the scikit-learn classification.



```
DESC:manner How did serfdom develop in and then leave Russia ?
ENTY:cremat What films featured the character Popeye Doyle ?
DESC:manner How can I find a list of celebrities ' real names ?
ENTY:animal What fowl grabs the spotlight after the Chinese Year of the Monkey ?
ABBR:exp What is the full form of .com ?
HUM:ind What contemptible scoundrel stole the cork from my lunch ?
HUM:gr What team did baseball 's St. Louis Browns become ?
HUM:title What is the oldest profession ?
DESC:def What are liver enzymes ?
HUM:ind Name the scar-faced bounty hunter of The Old West .
NUM:date When was Ozzy Osbourne born ?
DESC:reason Why do heavier objects travel downhill faster ?
HUM:ind Who was The Pride of the Yankees ?
HUM:ind Who killed Gandhi ?
ENTY:event What is considered the costliest disaster the insurance industry has ever faced ?
LOC:state What sprawling U.S. state boasts the most airports ?
DESC:desc What did the only repealed amendment to the U.S. Constitution deal with ?
NUM:count How many Jews were executed in concentration camps during WWII ?
DESC:def What is '' Nine Inch Nails '' ?
DESC:def What is an annotated bibliography ?
NUM:date What is the date of Boxing Day ?
ENTY:other What articles of clothing are tokens in Monopoly ?
HUM:ind Name 11 famous martyrs .
DESC:desc What 's the Olympic motto ?
DESC:desc What is the origin of the name ' Scarlett ' ?
ENTY:letter What 's the second-most-used vowel in English ?
HUM:ind Who was the inventor of silly putty ?
LOC:other What is the highest waterfall in the United States ?
ENTY:other Name a golf course in Myrtle Beach .
LOC:state Which two states enclose Chesapeake Bay ?
ABBR:exp What does the abbreviation AIDS stand for ?
ENTY:other What does a spermologer collect ?
NUM:count How many points make up a perfect fivepin bowling score ?
HUM:gr Which company that manufactures video-game hardware sells the '' super system '' ?
NUM:count How many Community Chest cards are there in Monopoly ?
DESC:desc What do Mormons believe ?
NUM:date When did the neanderthal man live ?
```

Figure 3. Sample training dataset train_5500.txt

8.3 Feature Extraction

Spark doesn't have direct support for nltk libraries. As Spark uses RDD, every feature extraction process is an output of a transformation. So we created a custom transformer for one of the feature extraction process which is POS tagging. Figure given below shows the workflow for the feature extraction process,

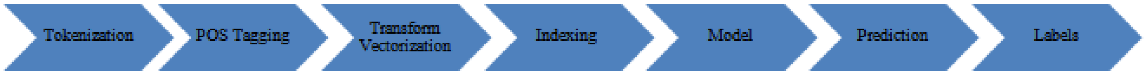


Figure Feature Extraction process for Spark Classifier

To initiate a Spark job, we first create its instance using SparkContext. We read the processed file which is stored in our local system. After reading the file, we get an RDD which holds the entire text file. To optimize the training time, we convert the RDD to a Spark dataframe using toDF() method. Databricks has done some benchmark testing to compare the performance of RDD and a dataframe [9]. The results clearly show that Dataframe is better in performance as compared to RDD. The following figure shows the performance results of the same,

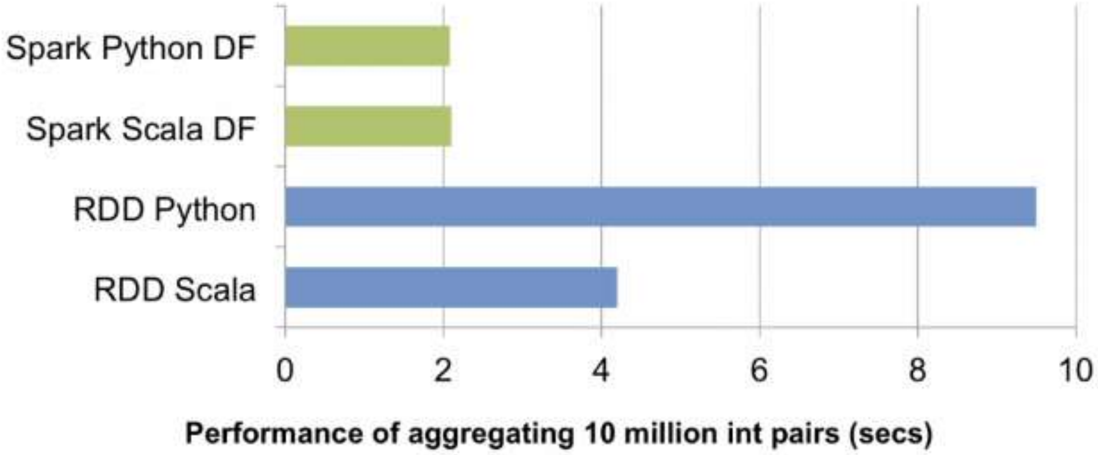
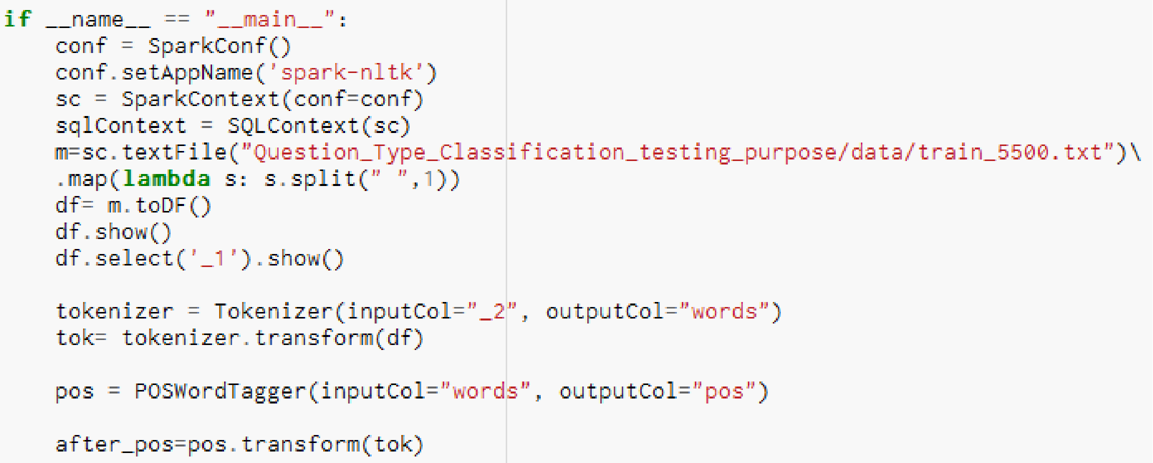


Figure Databricks performance results for RDD v/s Dataframe

The graph compares the aggregation results of 10M integer pairs on a single node during runtime. The difference between performance of Python and Scala dataframe operation is quite less as both generate JVM bytecode after execution and so nothing much to compare of. In case of RDD operations, the dataframe performance results beat both the language variants, the python implementation of RDD by a factor of 5 and Scala implementation of RDD by 2 [9].

1. Tokenization:

After reading the file into RDD, we convert it into a dataframe. Then we tokenize the data and pass it to the next step, POS tagging. The output of the tokenizer is saved in column "*words*".

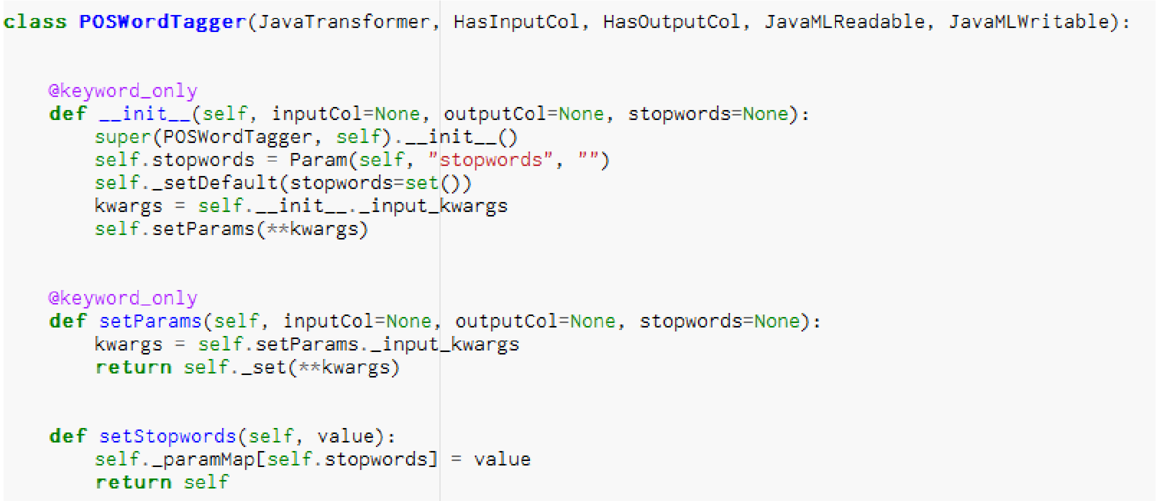


```
if __name__ == "__main__":
    conf = SparkConf()
    conf.setAppName('spark-nltk')
    sc = SparkContext(conf=conf)
    sqlContext = SQLContext(sc)
    m=sc.textFile("Question_Type_Classification_testing_purpose/data/train_5500.txt")\
    .map(lambda s: s.split(" ",1))
    df= m.toDF()
    df.show()
    df.select('_1').show()

    tokenizer = Tokenizer(inputCol="_2", outputCol="words")
    tok= tokenizer.transform(df)

    pos = POSWordTagger(inputCol="words", outputCol="pos")

    after_pos=pos.transform(tok)
```

Figure Tokenization and POS tagging

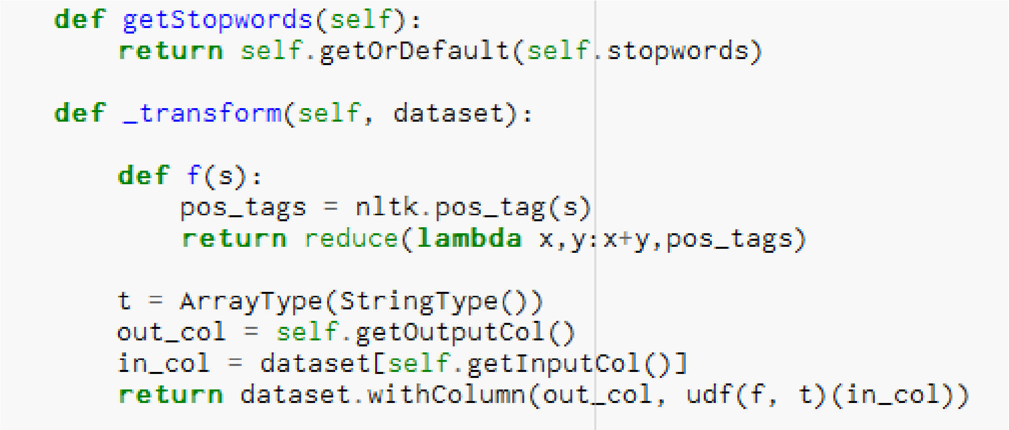
2. POS tagging:

Output of tokenizer, "*words*" is given as input column to the custom transformer. As Spark doesn't support the nltk libraries directly, we wrote a custom transformer.



```
class POSWordTagger(JavaTransformer, HasInputCol, HasOutputCol, JavaMLReadable, JavaMLWritable):


    @keyword_only
    def __init__(self, inputCol=None, outputCol=None, stopwords=None):
        super(POSWordTagger, self).__init__()
        self.stopwords = Param(self, "stopwords", "")
        self._setDefault(stopwords=set())
        kwargs = self.__init__._input_kwargs
        self.setParams(**kwargs)


    @keyword_only
    def setParams(self, inputCol=None, outputCol=None, stopwords=None):
        kwargs = self.setParams._input_kwargs
        return self._set(**kwargs)


    def setStopwords(self, value):
        self._paramMap[self.stopwords] = value
        return self
```

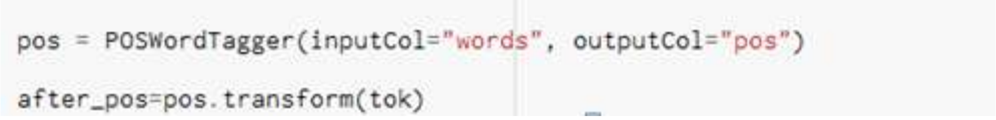
Figure Custom transformer POSWordTagger



```
    def getStopwords(self):
        return self.getOrDefault(self.stopwords)

    def _transform(self, dataset):

        def f(s):
            pos_tags = nltk.pos_tag(s)
            return reduce(lambda x,y:x+y,pos_tags)

        t = ArrayType(StringType())
        out_col = self.getOutputCol()
        in_col = dataset[self.getInputCol()]
        return dataset.withColumn(out_col, udf(f, t)(in_col))
```

Figure Transform function implementation

The transform method uses the *pos_tags()* method from the nltk library to extract the POS tag for each token. The output is saved in the column "*pos* "



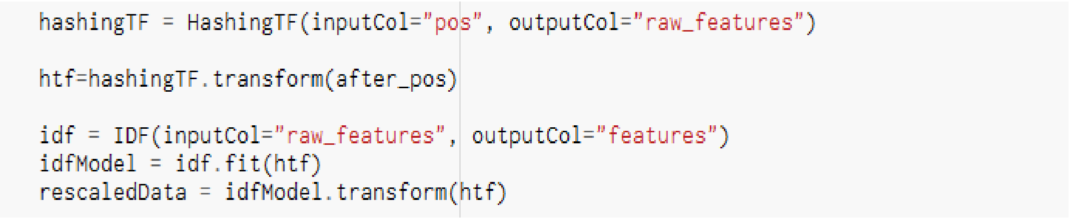
```
pos = POSWordTagger(inputCol="words", outputCol="pos")

after_pos=pos.transform(tok)
```

Figure Calling POSWordTagger

3. Tf-idf vectorizer:

Output of pos tagger, "*pos*" is given as input column to the custom transformer.

The output of POS tagger transformer is used to extract the tf-idf vectors. The output of tf-idf vectorizer is saved in the column "*raw_features*".

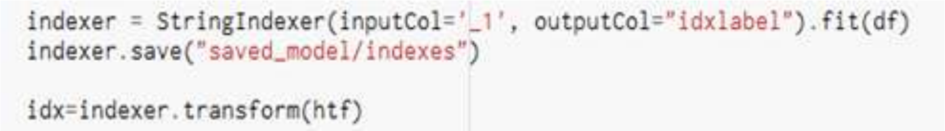


```
hashingTF = HashingTF(inputCol="pos", outputCol="raw_features")

htf=hashingTF.transform(after_pos)

idf = IDF(inputCol="raw_features", outputCol="features")
idfModel = idf.fit(htf)
rescaledData = idfModel.transform(htf)
```

Figure Extracting tf-idf vectors

4. Indexing:

Spark doesn't takes string values as labels. So we convert each label to a numeric value by using the *StringIndexer()* method. The output of the indexer consist of numeric values for each label. The transformed indexer is saved for future use as we need to retrieve the original label back. The output of the indexer is saved in the column "*idxlabel*"

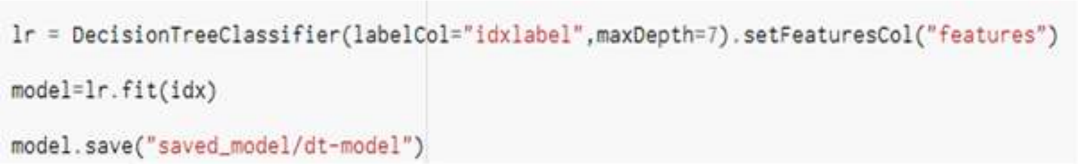


```
indexer = StringIndexer(inputCol='_1', outputCol="idxlabel").fit(df)
indexer.save("saved_model/indexes")

idx=indexer.transform(htf)
```

Figure Extracting tf-idf vectors

8.4 Predictive Modeling

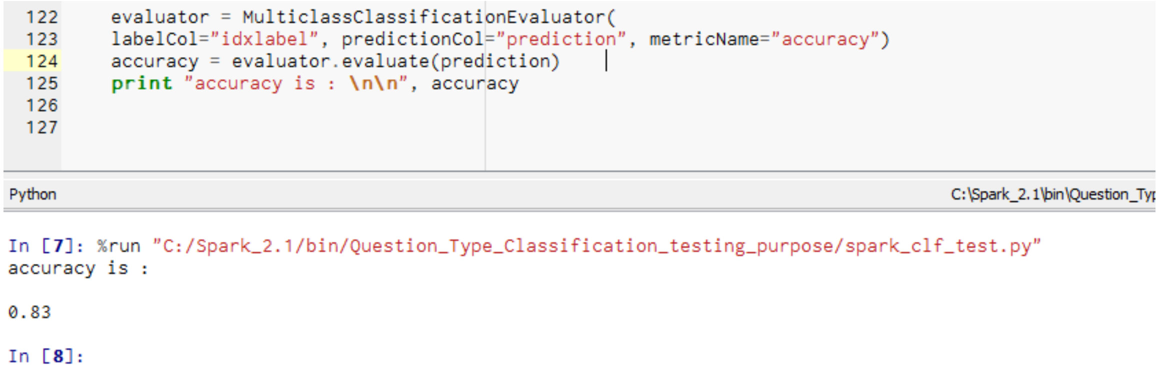
After we extract all the features, we build our model using decision tree classifier. The input to the classifier is the final set of features obtained after tf-idf transformation and the label column has been given the output of the indexer



```
lr = DecisionTreeClassifier(labelCol="idxlabel",maxDepth=7).setFeaturesCol("features")

model=lr.fit(idx)

model.save("saved_model/dt-model")
```

Figure Extracting tf-idf vectors

After building the model, when we evaluate the model using the test dataset, we get an accuracy of 83%.



```
evaluator = MulticlassClassificationEvaluator(
labelCol="idxlabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(prediction)
print "accuracy is : \n\n", accuracy
```

```
In [7]: %run "C:/Spark_2.1/bin/Question_Type_Classification_testing_purpose/spark_clf_test.py"
accuracy is :

0.83

In [8]:
```

Figure Spark Model Evaluation

The model is saved to disk to use it for later purpose (Spark Streaming).

9. Spark Streaming

The aim was to create a scalable application with notable performance gain. So in order to simulate a real-world scenario, we created a Spark streaming application. We used the Spark Streaming API that is available in the Spark stack to create a streaming application. The figure given below shows the architecture of Spark Streaming [10],

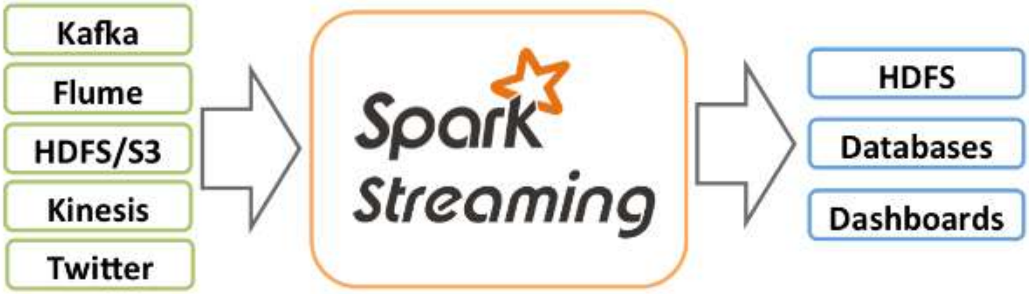


Figure Spark Streaming architecture

Input sources can be any streaming service (producer), which will generate the data and submit it to the Spark streaming application which will act as a consumer. After the data has been ingested and the required operation has been performed on the data, the output can be written or saved to any storage services. We can also create a dashboard which can show the impact and behavior of all the processing in the form of graphs.

For our application, we are using Amazon Kinesis as the producer. So now, instead of providing some sample questions for prediction, we give a stream of questions to the classifier.

9.1 Amazon Kinesis Streams

Amazon Kinesis streams is one of the many services that is provided by Amazon Web Services (AWS). It is used to create streaming data. As the data is generated continuously from multiple sources, the size of data to be processed per hour has reached upto TBs. Amazon Kinesis helps to collect such huge data and provide ways to process and store it at very low cost.

Advantages of using Amazon Kinesis are,

1. Real-time streaming- We can collect the data as and when it is generated and we can perform any analysis or computation in real time.
2. Parallel processing- We can process the same Kinesis stream by running multiple Kinesis applications concurrently.
3. Scalability- Kinesis streams can scale to handle Megabytes or even Terabytes of data per hour. We can adjust the throughput required for our application dynamically.
4. Low processing cost- It is cheap to use Amazon Kinesis streams with the rates being $0.015 per hour.
5. Reliability- It replicates the data across three Amazon Region facilities and it can store the data for seven days, thereby avoiding any information loss due to failure.

9.1.1 Amazon Kinesis Streams Architecture

The following figure shows the architecture of Kinesis Streams [11],

Ama

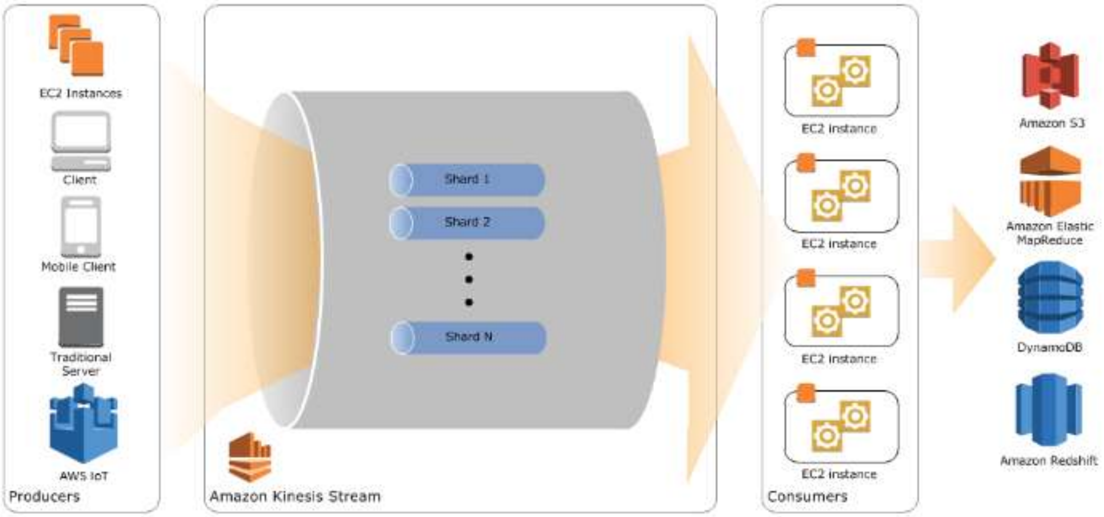


Figure High level architecture of Kinesis Streams

1. Producers- Any source that can generate data

2. Kinesis Application- EC2 running application that acts as a consumer for the streams of data. Output of the consumer can be an input to another Kinesis application running on EC2 instance or we can store the output on any storage services.

3. Shards- It distinctly identifies data records in a stream. A stream consists of atleast one shard. We can dynamically increase or decrease the number of shards with an increase or decrease in data rate.

4. Partition- Data records are segregated into shards using a partition key. When data is regrouped, this key helps in identifying that the particular record belongs to which shard.

5. Amazon Kinesis Client Library (KCL)- Applications are build using KCL and are used to process the data fetched from the streams.

9.1.2 Prerequisites

In order to use AWS services, we need to do following steps,

1. Create a login account.

2. Install the AWS CLI tool. We can use the following command to install the CLI,

```
$ pip install awscli
$ sudo pip install --upgrade awscli
```

3. Set up our AWS credentials in our environment. It includes two variables to set,

```
aws_access_key_id = YOUR_KEY
aws_secret_access_key = YOUR_SECRET
```

We use to following command to setup the environment,

```
$ aws configure
```

The credentials are saved in a "credentials" file which is located in the directory where we have installed the AWS. Now our environment is ready to communicate with any AWS services.

4. Install boto3- It is the AWS SDK build for Python. It provides a set of libraries that we can use to communicate with AWS services. Here we will use boto3 to create a Kinesis Client. We use the following command to install boto3,

```
$ pip install boto3
```

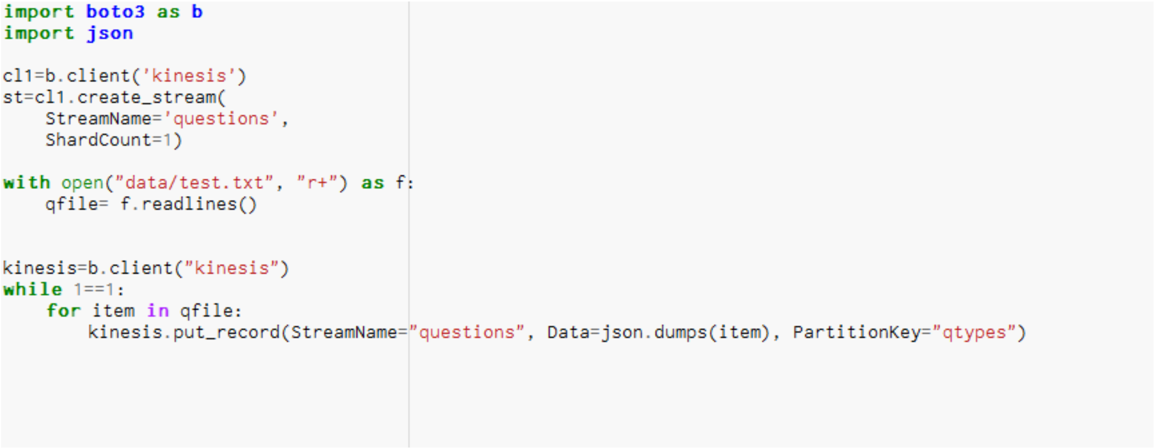
5. Choose a region - The region name is default which is us-west-2. We can change the region name by changing the value of the *region* field in the aws config file.

The next step is to create a Kinesis Producer which will create a Kinesis stream on AWS. The stream will comprise of different questions which are read from a text file.

9.1.3 Create a Kinesis Producer

We create a Kinesis client to create and submit out streams on AWS. As a data source, we are using a dataset which comprises of 200 unique questions. We create a stream with the name as "**questions**" and the partition key as "**qtypes**". The stream need an input in the JSON format and so we use the *json.dumps()* method while putting the records using *kinesis.put_record()*. Default shard count is 1.

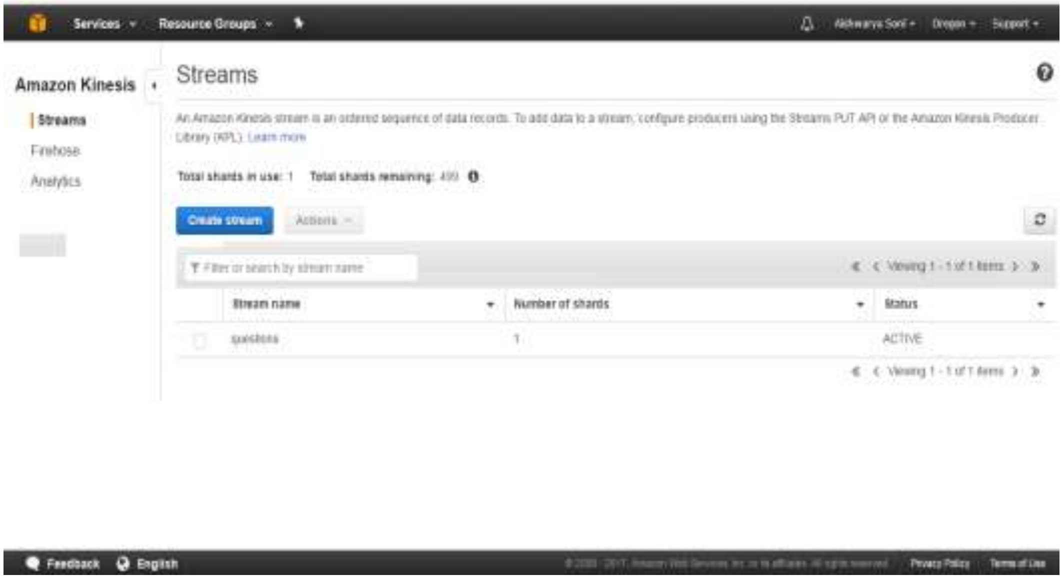
As shown in the figure below, the Kinesis stream client will continuously submit the questions to the stream using an infinite loop,



```
import boto3 as b
import json

cl1=b.client('kinesis')
st=cl1.create_stream(
    StreamName='questions',
    ShardCount=1)

with open("data/test.txt", "r+") as f:
    qfile= f.readlines()


kinesis=b.client("kinesis")
while 1==1:
    for item in qfile:
        kinesis.put_record(StreamName="questions", Data=json.dumps(item), PartitionKey="qtypes")
```

Figure Kinesis Producer

In the first figure given below we can see the "**questions**" stream that is created on AWS. In the second figure, we can monitor the behavior of the system when we start putting the records on the stream. These graphs are generated by the CloudWatch monitoring system that is built-in in AWS.



Streams

An Amazon Kinesis stream is an ordered sequence of data records. To add data to a stream, configure producers using the Streams PUT API or the Amazon Kinesis Producer Library (KPL). Learn more

Total shards in use: 1 Total shards remaining: 499

Filter or search by stream name

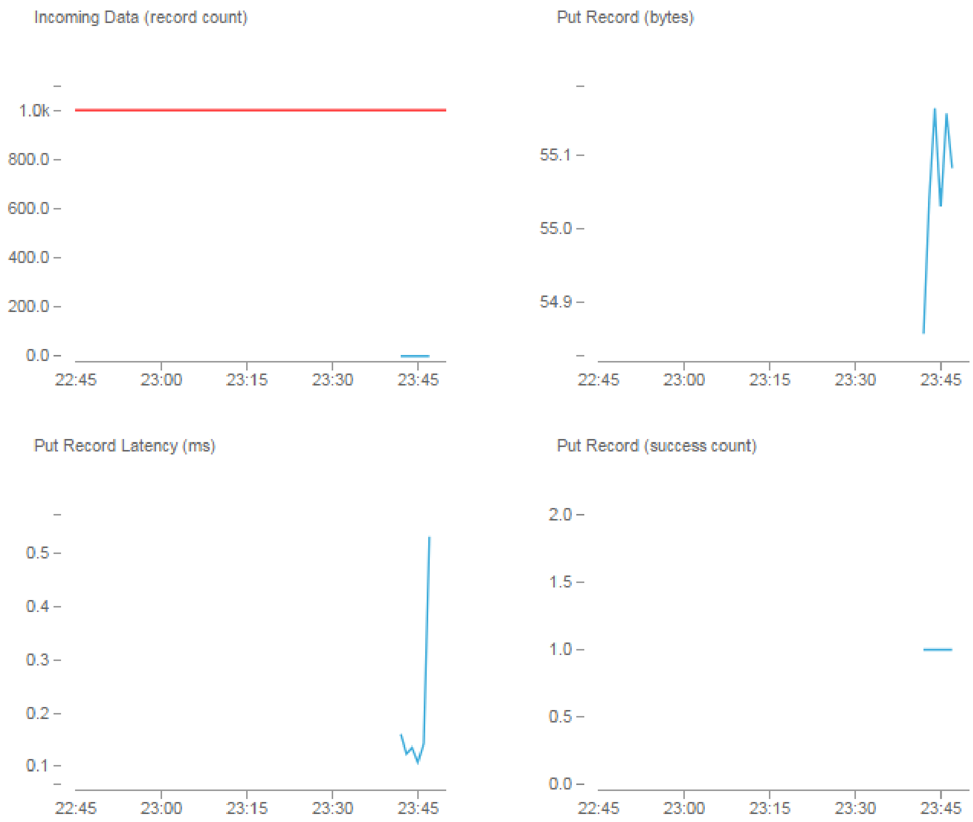
Viewing 1 - 1 of 1 items

	Stream name	Number of shards	Status
□	questions	1	ACTIVE

Viewing 1 - 1 of 1 items

© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Figure Stream generated on AWS


Incoming Data (record count)

Put Record (bytes)

Put Record Latency (ms)

Put Record (success count)

Figure Monitoring system on AWS Kinesis Stream

9.2 Create a Spark Streaming Consumer

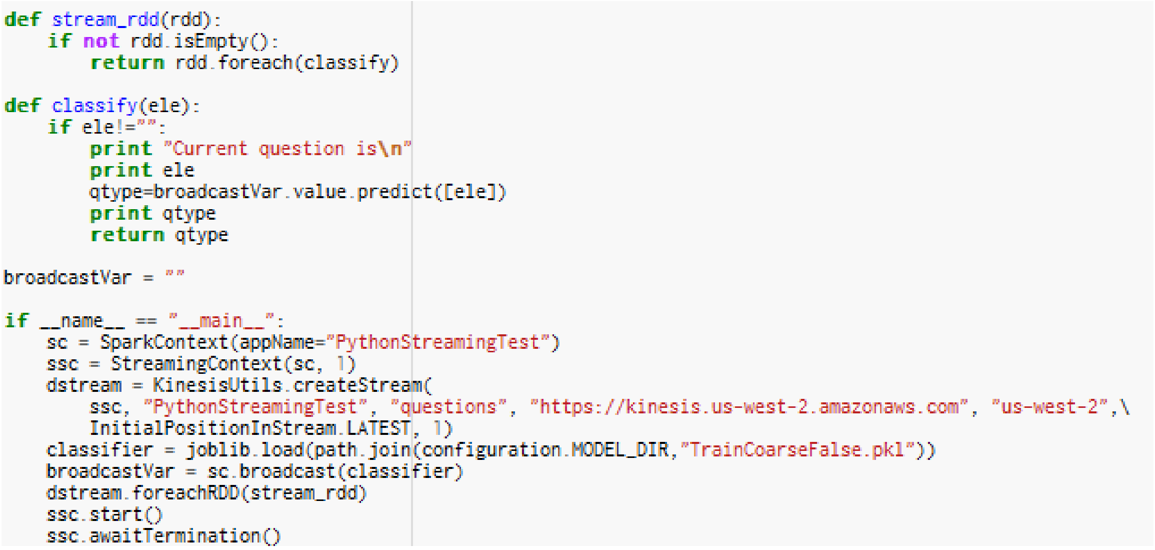
In order to fetch the data produced by Kinesis, we wrote a Spark streaming consumer application. We wanted to observe how the two classifiers behave in real-time streaming. So we built two streaming applications in which we used the scikit-learn classifier and the spark classifier.

We use the PySpark library to implement any Spark job in Python. To initiate a Spark job, we use the *SparkContext()* object which is imported from the PySpark library. It is the starting point for any Spark application. To initiate spark streaming, we create a *StreamingContext()* object which is imported from the spark streaming library.

9.2.1 Implementing Spark Consumer Using scikit-learn Classifier

We use the concept of broadcast variables in both the application. We load the stored classifier from the disk and give it to the broadcast variable. When we do *sc.broadcast()*, we broadcast the classifier to all the worker nodes. So when the stream of questions comes, we have the classifier available on all the workers.

The figure given below gives the implementation snippet for the spark consumer application using scikit learn classifier. As shown in the figure, we use the Kinesis Utils library to consume the Kinesis streams. The parameter *InitialPositionStream.LATEST* indicates that we are interested in the latest streams. Shard count is set to 1 as we are using only 1 shard. We deploy our classifier when we do *broadcastVar.value* and then we call the *predict()* method of the classifier for each question.



```
def stream_rdd(rdd):
    if not rdd.isEmpty():
        return rdd.foreach(classify)

def classify(ele):
    if ele!="":
        print "Current question is\n"
        print ele
        qtype=broadcastVar.value.predict([ele])
        print qtype
        return qtype

broadcastVar = ""

if __name__ == "__main__":
    sc = SparkContext(appName="PythonStreamingTest")
    ssc = StreamingContext(sc, 1)
    dstream = KinesisUtils.createStream(
        ssc, "PythonStreamingTest", "questions", "https://kinesis.us-west-2.amazonaws.com", "us-west-2",\
        InitialPositionInStream.LATEST, 1)
    classifier = joblib.load(path.join(configuration.MODEL_DIR,"TrainCoarseFalse.pkl"))
    broadcastVar = sc.broadcast(classifier)
    dstream.foreachRDD(stream_rdd)
    ssc.start()
    ssc.awaitTermination()
```

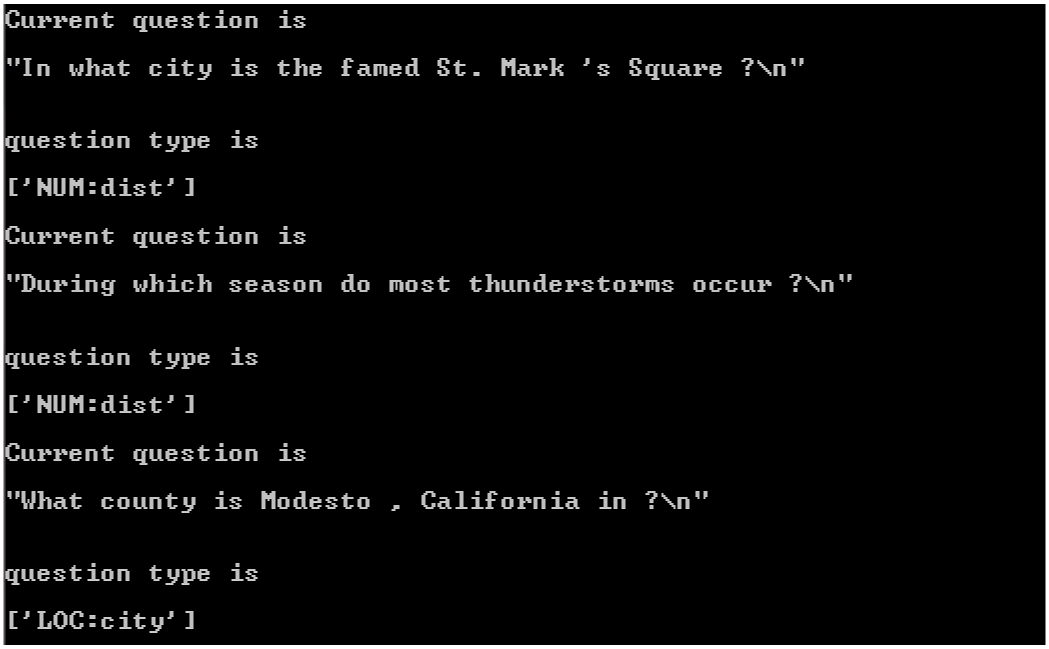
Figure Spark consumer application using scikit-learn classifier

We use following command to run the spark job.

"*spark-submit --driver-memory 5g Question_Type_Classification/spark-stream.py*"

As we are running spark on a local machine, we give the driver memory a value (here 5g) to indicate what amount of RAM is to be given to the executors on each worker node. We cannot set the executor memory when we run spark on a local machine. The reason being that, when we run spark on a local machine, the worker resides insides the JVM process of the driver. The default memory is set to 512MB. So we have to reset it to a higher value depending upon available system configuration. When we run Spark in a cluster mode, then we can set the executor memory to some value as we would have a different master (other than the driver daemon) which can be Mesos or Yarn or Hadoop itself.

The output of the spark streaming is given below,



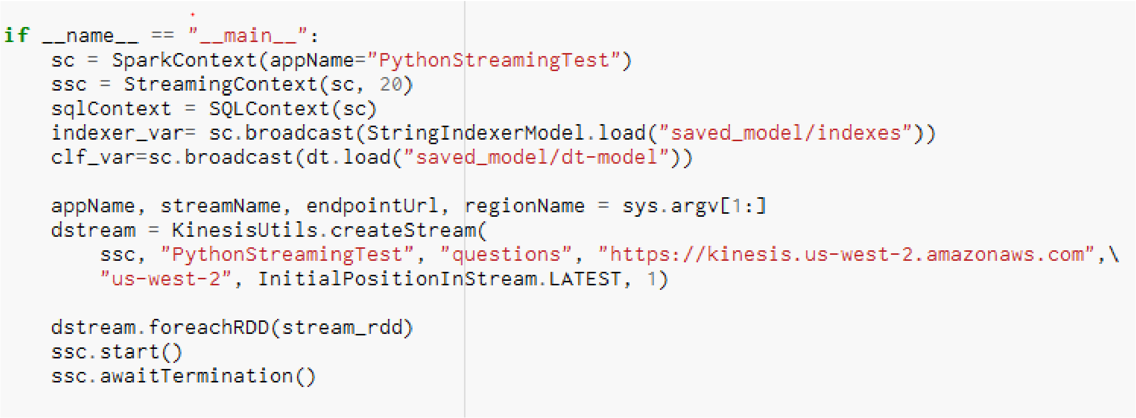
```
Current question is
"In what city is the famed St. Mark ’s Square ?\n"

question type is
[’NUM:dist’]
Current question is
"During which season do most thunderstorms occur ?\n"

question type is
[’NUM:dist’]
Current question is
"What county is Modesto , California in ?\n"

question type is
[’LOC:city’]
```

Figure Spark streaming output

9.2.2 Implementing Spark Consumer Using Spark Classifier

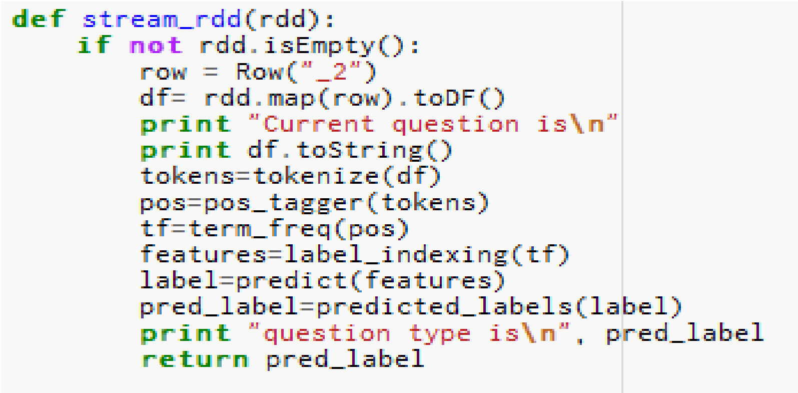
In this, we broadcast the two models from the driver program. First is the indexer that we saved in the previous spark program while training the classifier as we have to retrieve the original labels from the indexed one and second is the classifier itself. For each stream of questions, we extract the features and give it to the classifier for prediction.

The following snippets show the implementation of the spark consumer using the spark classifier,



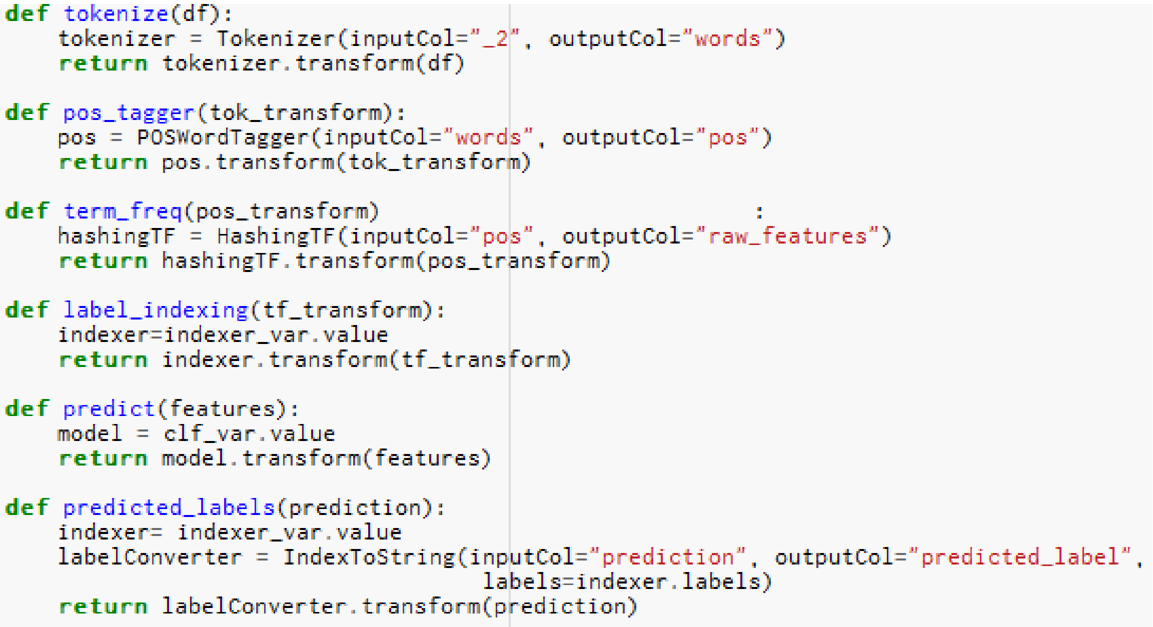
```
if __name__ == "__main__":
    sc = SparkContext(appName="PythonStreamingTest")
    ssc = StreamingContext(sc, 20)
    sqlContext = SQLContext(sc)
    indexer_var= sc.broadcast(StringIndexerModel.load("saved_model/indexes"))
    clf_var=sc.broadcast(dt.load("saved_model/dt-model"))

    appName, streamName, endpointUrl, regionName = sys.argv[1:]
    dstream = KinesisUtils.createStream(
        ssc, "PythonStreamingTest", "questions", "https://kinesis.us-west-2.amazonaws.com",\
        "us-west-2", InitialPositionInStream.LATEST, 1)

    dstream.foreachRDD(stream_rdd)
    ssc.start()
    ssc.awaitTermination()
```

Figure Spark driver program



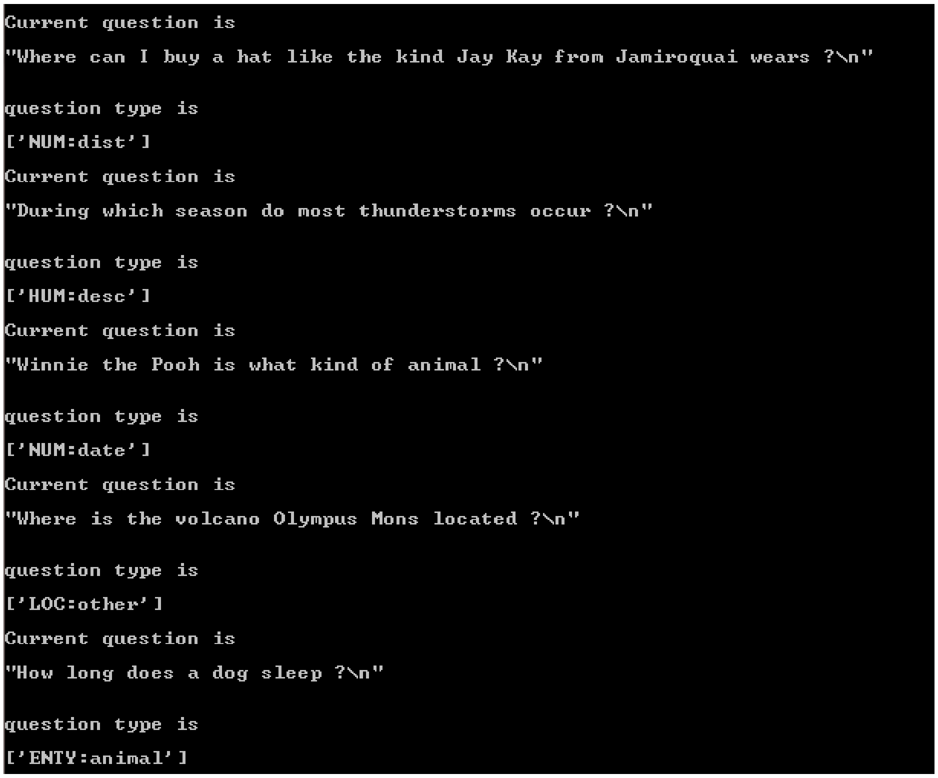
```
def stream_rdd(rdd):
    if not rdd.isEmpty():
        row = Row("_2")
        df= rdd.map(row).toDF()
        print "Current question is\n"
        print df.toString()
        tokens=tokenize(df)
        pos=pos_tagger(tokens)
        tf=term_freq(pos)
        features=label_indexing(tf)
        label=predict(features)
        pred_label=predicted_labels(label)
        print "question type is\n", pred_label
        return pred_label
```

Figure stream_rdd() function implementation



```
def tokenize(df):
    tokenizer = Tokenizer(inputCol="_2", outputCol="words")
    return tokenizer.transform(df)

def pos_tagger(tok_transform):
    pos = POSWordTagger(inputCol="words", outputCol="pos")
    return pos.transform(tok_transform)

def term_freq(pos_transform)                                  :
    hashingTF = HashingTF(inputCol="pos", outputCol="raw_features")
    return hashingTF.transform(pos_transform)

def label_indexing(tf_transform):
    indexer=indexer_var.value
    return indexer.transform(tf_transform)

def predict(features):
    model = clf_var.value
    return model.transform(features)

def predicted_labels(prediction):
    indexer= indexer_var.value
    labelConverter = IndexToString(inputCol="prediction", outputCol="predicted_label",
                                   labels=indexer.labels)
    return labelConverter.transform(prediction)
```

Figure Feature extraction and prediction



```
Current question is
"Where can I buy a hat like the kind Jay Kay from Jamiroquai wears ?\n"

question type is
['NUM:dist']
Current question is
"During which season do most thunderstorms occur ?\n"

question type is
['HUM:desc']
Current question is
"Winnie the Pooh is what kind of animal ?\n"

question type is
['NUM:date']
Current question is
"Where is the volcano Olympus Mons located ?\n"

question type is
['LOC:other']
Current question is
"How long does a dog sleep ?\n"

question type is
['ENTY:animal']
```

Figure Output of the spark consumer

10. Performance Improvements

Due to resource crunch on a local machine, there is not much we can do to improve the performance. The real power of spark can only be achieved if we have the computation power. So to improve the performance of the application, we deployed our application on AWS EC2 instances. EC2 provides us instances with the computing resources which we can choose according to our needs.

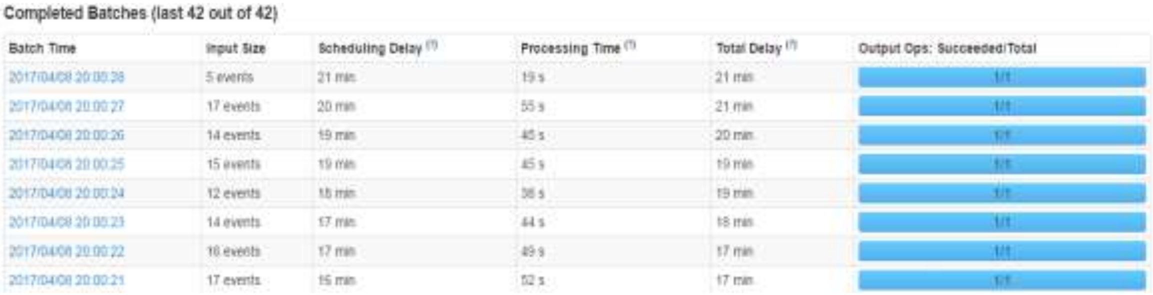
The advantages of using EC2 are,

1. Easy scalability- We can increase or decrease computing resources at will and in few minutes. We can run thousands of instances without worrying about how to manage all the instance as AWS does it for us. Application can scale up or down at runtime depending upon the need.
2. User Control- Users have a full control on each instance. A user can start, stop or restart any instance when required. User has the access to API's or console to manage the instances.
3. Reliable- According to AWS, its cloud is 99.95% reliable. So even if any instance gets failed or crashed due to some reason, another instance is brought up automatically without the customer to realize it and facing any downtime.
4. Secure- AWS allows us to create Virtual Private Cloud (VPC) so that we can have a personalized access to our instances.
5. Inexpensive- The cost of an instance is quite cheap. It works on the principle of pay per use.

AWS provides different types of instances which can be chosen as per the use case. For our use case, we are using instances of the M3 family. It provides us with good computation resources that is perfect for most of the applications. We are using a 3 node cluster and each node is an m3.xlarge instance. Each instance has 4 vCPU, 15 GB of RAM and 2x40 GB of SSD storage.

When we run the scikit learn classifier on a local machine, the performance was not great. The reason being that, it has to fetch the NER jar for each token for each stream of question , tag each word and bring back the results. This degraded the performance of the application. So we thought to create a new classifier with all the features except NER tagging. We call this classifier as "scikit lite" classifier. So we had tf-idf vectors, POS tagging and semantic analysis as the features for this new classifier. The net result was increase in performance but the accuracy of the scikit lite classifier was only 68%. When we run both the applications individually on the spark framework we can notably see the performance change.

To evaluate the performance, we compare the throughput of both the classifiers at different window size. The window size will signify how much seconds the spark application will wait to take the next batch of input records. The window size can be set to any value, but to test the performance on the local machine we took two window sizes, 1 and 15. We compared the results of 8 batches of each classifier application for each window size.

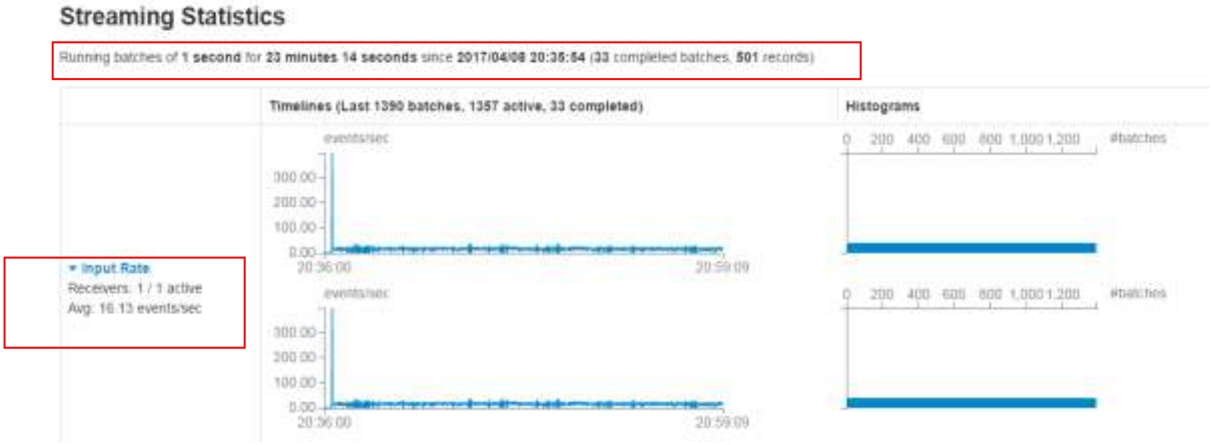


Completed Batches (last 42 out of 42)

Batch Time	Input Size	Scheduling Delay (?)	Processing Time (?)	Total Delay (?)	Output Ops: Succeeded/Total
2017/04/08 20:00:38	5 events	21 min	19 s	21 min	1/1
2017/04/08 20:00:27	17 events	20 min	55 s	21 min	1/1
2017/04/08 20:00:26	14 events	19 min	45 s	20 min	1/1
2017/04/08 20:00:25	15 events	19 min	45 s	19 min	1/1
2017/04/08 20:00:24	12 events	18 min	38 s	19 min	1/1
2017/04/08 20:00:23	14 events	17 min	44 s	18 min	1/1
2017/04/08 20:00:22	16 events	17 min	49 s	17 min	1/1
2017/04/08 20:00:21	17 events	16 min	52 s	17 min	1/1

Figure Sample output of each completed batches

The following graphs show us the performance results for the fully loaded classifier (old classifier with all the features) and the scikit lite classifier,



Streaming Statistics

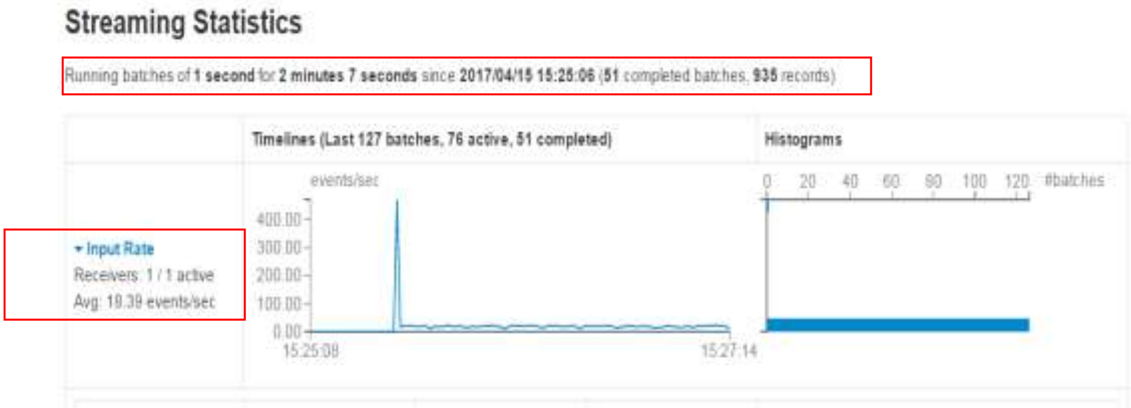
Running batches of 1 second for 23 minutes 14 seconds since 2017/04/08 20:35:54 (33 completed batches, 501 records)

Timelines (Last 1390 batches, 1357 active, 33 completed)

Histograms

▾ Input Rate
Receivers: 1 / 1 active
Avg: 16.13 events/sec

Figure Old classifier stream performance with window size 1:



Streaming Statistics

Running batches of 1 second for 2 minutes 7 seconds since 2017/04/15 15:25:06 (51 completed batches, 935 records)

Timelines (Last 127 batches, 76 active, 51 completed)

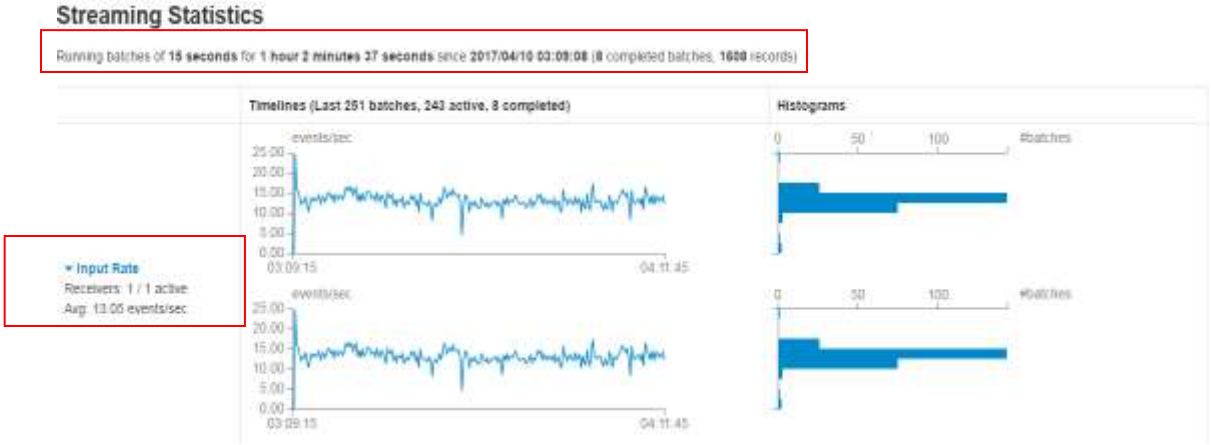
Histograms

▾ Input Rate
Receivers: 1 / 1 active
Avg: 19.39 events/sec

Figure New classifier stream performance with window size 1:

As observed from the above graphs, the time taken to process the records is less as compared to the old classifier. Total number of records processed is also high in the new classifier. When we compare the throughput of 8 batches of each classifier application, the processing performance for the old classifier is 0.53 records/second and for the new classifier we get 7.48 records/second.

When we change the window size to 15, we get the following graphs,



Streaming Statistics

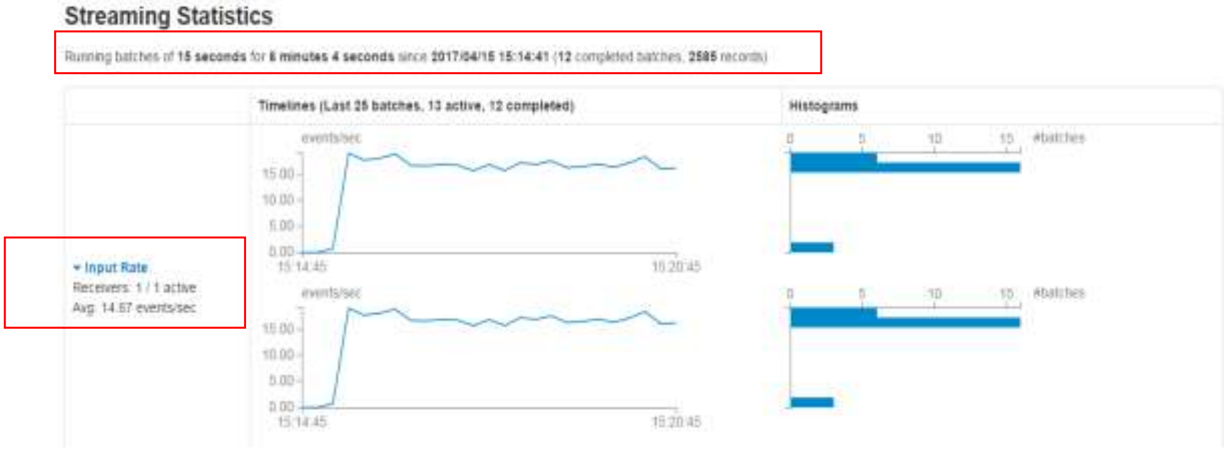
Running batches of **15 seconds** for **1 hour 2 minutes 37 seconds** since **2017/04/10 03:09:08** (**8** completed batches, **1608** records)

Timelines (Last 251 batches, 243 active, 8 completed)

Histograms

▾ Input Rate
Receivers: 1 / 1 active
Avg: 13.05 events/sec

Figure Old classifier stream performance with window size 15:



Streaming Statistics

Running batches of **15 seconds** for **6 minutes 4 seconds** since **2017/04/15 15:14:41** (**12** completed batches, **2585** records)

Timelines (Last 25 batches, 13 active, 12 completed)

Histograms

▾ Input Rate
Receivers: 1 / 1 active
Avg: 14.67 events/sec

Figure New classifier stream performance with window size 15:

As observed from the above graphs, the time taken to process the records is less as compared to the old classifier. Total number of records processed is also high in the new classifier. When we compare the throughput for 8 batches of each classifier application, the processing performance for the old classifier is 0.42 records/second and for the new classifier we get 7.95 records/second. So, for window size 15, the performance for old classifier degraded but there was a slight improvement in performance for the new classifier. We summarize the observations in the following table,

Observations	**Old classifier application**	**New classifier application**
Accuracy (%)	93	68
Best Performance	0.53	7.95
Overall Performance	Bad	Good
Best window size	1	15

Table 1. Observation table for old v/s new classifier application

From the above table, it is clear that the new classifier performs better than the old classifier. As the goal of the use case was to increase the performance of the application, we chose the new classifier, i.e. the scikit lite classifier to deploy on the AWS EC2 instances. So we deploy our new classifier and the classifier written using spark MLlib on the 3 node cluster to observe and compare the performance results on different loads.

10.1 EC2 Performance Observations

We deployed the two applications on the AWS EC2 cluster to evaluate the performance and scalability. In order to execute our application, we setup the worker nodes with all the dependencies that was needed to run the application as per the guidelines on AWS [12].

To evaluate the performance we tested our application by giving low, medium and high load. We also changed the window size which varies to 1, 5, 10, 15 to find the optimum performance. After testing the application on various parameters, we drew the performance graphs using Tableau. The graph has been divided into two sections, first one is average number of records processed per sec and second section is the average input size at different window size. Records processed per sec for each window is obtained by taking the average number of records obtained in 10 batches and average processing time for each batch.

10.1.1 Performance Evaluation for Scikit lite Spark Streaming Application

To begin with, we first evaluated how our scikit lite spark streaming application performed on EC2 cluster and what are the notable observations. The following graph shows the performance results on EC2 cluster,

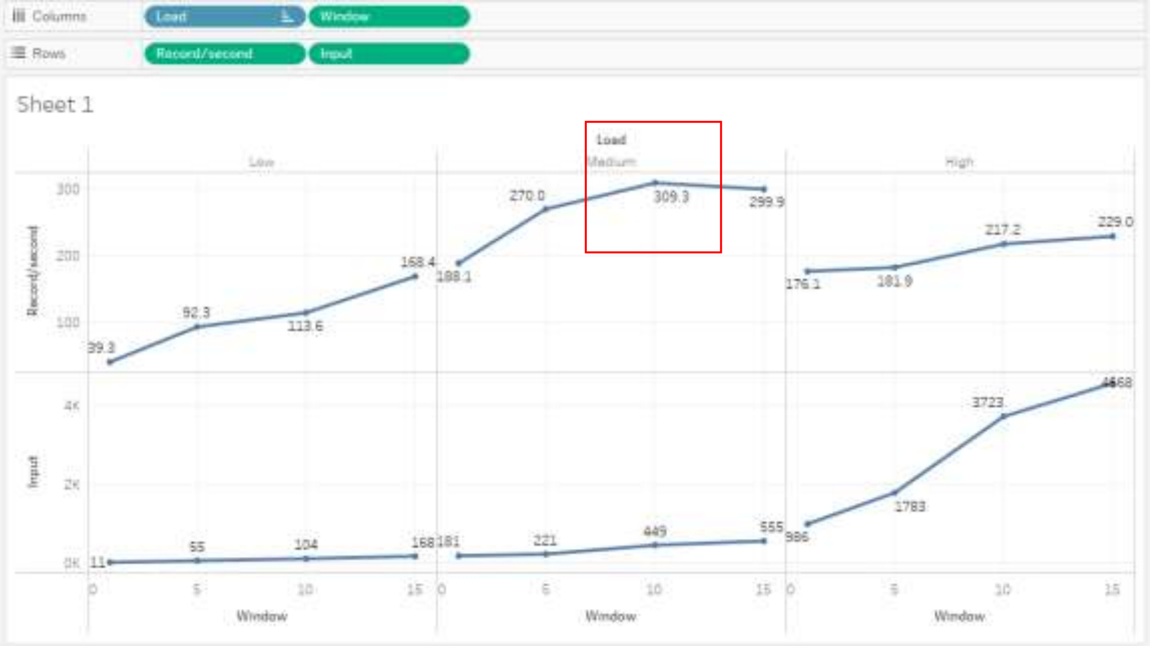


Figure EC2 performance results for scikit lite spark application

As observed from above graph, for low and high load the application performs better as the load increases. For medium load, at window 10, we get the maximum records processed as compared to other load setting and window size. After that point, the record processing decreases. Also, that is the best point where we get maximum record processing. The conclusion derived from above observations was in order to achieve the best performance, we need to set window size to 10 and load should be medium.

10.1.2 Performance Evaluation for Spark Classifier Streaming Application

We evaluated or spark classifier streaming application on similar parameters on the EC2 cluster. The following graph shows the performance results on EC2 cluster,

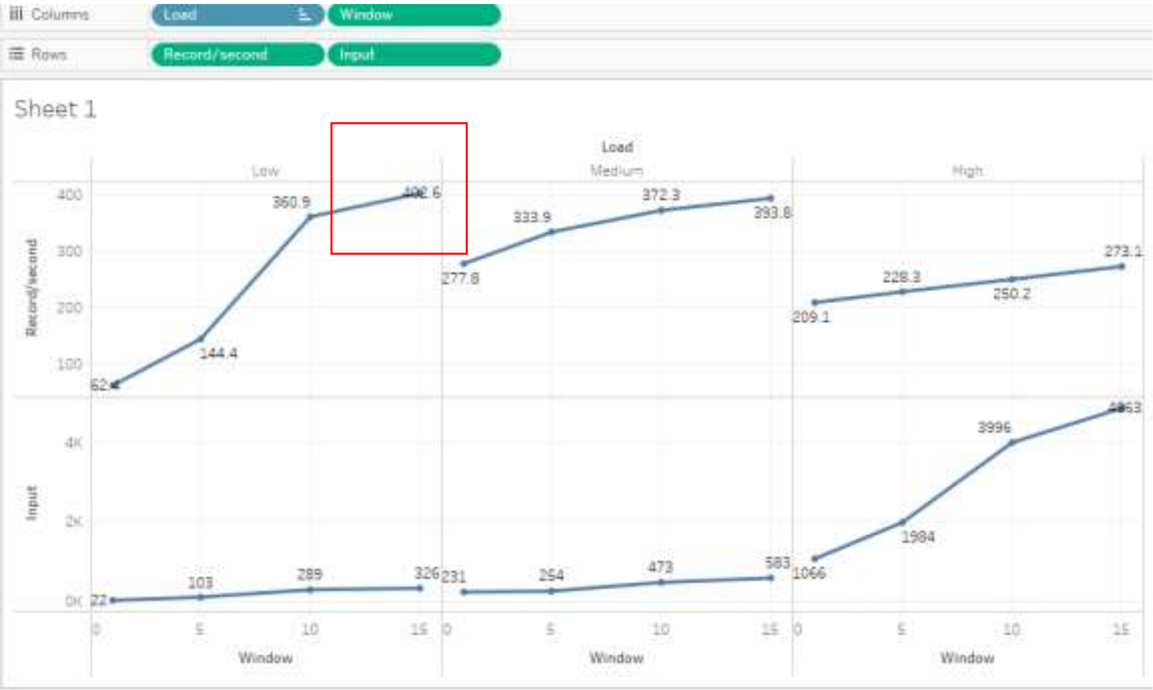


Figure EC2 performance results for spark classifier streaming application

For the above spark application we observed that it behaves better as the load increases. There is no definite good point to set to achieve optimum performance. The window size and load can be set according to the requirement of the use case. The trade off criteria can given as,

window size ∝ delay in response.

More the window size to get the performance, more is the response time for the request. For the current scenario, low load and window size 15 seems to be a sweet spot as we get more throughput then any point at the graph.

The optimum points obtained from the performance analysis of both the applications can be summarized in the following,

Observations	**Scikit lite classifier application**	**Spark classifier application**
Accuracy (%)	68	83
Best Performance	309.3	402.6
Window size	10	15
Load	Medium	Low

Table 2. Observation table for scikit lite v/s spark classifier application

11. Conclusion and Future Work

In this project, we created an application that would classify any question to its question type. During the project, we created three applications which includes the full loaded classifier, scikit lite classifier and the spark classifier. We performed some performance testing on local machine which didn't helped much. So we deployed the lite classifier application and the spark classifier application on AWS EC2 cluster. We tested our application on different parameters and came up with a set of observations.

In conclusion of the observations we can say that,

1. Spark classifier application performs better than the scikit lite classifier application.
2. We get less latency (delay in response) in the scikit lite classifier application as the window size is less than the spark classifier application.
3. If the focus is to get more input records to process in short period of time and accuracy is not an issue, then we can go ahead with the scikit lite classifier application. But this case is very unlikely as many applications will chose the first option as they are getting good accuracy and better throughput. Latency can be compromised for better results.

Finally while doing the project, we also observed that by enhancing the model with the semantic features improves the classification accuracy. Also, "*What*" type of questions are very difficult to classify, for example, consider this question *"What is the PH Scale?"*. This question can be classified as a numeric value or a definition. So proper modeling is required to classify "*What*" type of questions.

In accuracy, scikit learn beats spark as scikit learn is more tuned to perform classification and is more matured then spark. In Performance testing, the spark classifier performs better on every level (apart from latency) as compared to scikit learn classifier. Selecting the performance v/s accuracy tradeoff depends totally upon the requirement of the application.

In future work, we can say that the current work can be extended in multiple ways. We can apply some deep semantic techniques for semantic analysis. Stanford is currently working on how can we apply deep learning in natural language processing [13]. We can use some artificial intelligence techniques that enables learning for new questions and question types while performing classification. We can assign multiple labels to a single question to diversify the classification categories. In this project, we have the dataset that assumes unique label for each question. This may lead to some misclassification of certain types of questions, especially in "*What*" type of questions. By having multiple possible labels for a question, we can improve the accuracy of the model. We can also have a dedicated classifier to classify "*What*" type of questions. This will help in defining a more accurate model with better training and features set which will only classify "*What*" type of questions. Lastly, we can have dynamic selection of classifier for certain type of questions. So depending upon the keywords such as "*Who, What, When, Where, Why, How*", we can select the appropriate classifier during runtime. This will help to provide more accurate results as the classifier is only trained to predict the question type for that particular type of keyword.

12. References

[1] Loni B (2011) A survey of state-of-the-art methods on question classification. Delft University of Technology, Delft, pp 1–40

[2] Mollá, Diego, and José Luis Vicedo. "Question answering in restricted domains: An overview." Computational Linguistics 33.1 (2007): 41-61.

[3] Li, D. Roth. 2002. "Learning Question Classifiers". In Proceedings of ACL 2002.

[4] "TREC website", http://trec.nist.gov/data.html

[5] T. Joachims. "Text categorization with support vector machines: Learning with many relevant features". In Claire N´edellec and C´eline Rouveirol, editors, Proceedings of the European Conference on Machine Learning, pages 137–142, Berlin, 1998. Springer.

[6] "Botkit framework", https://github.com/howdyai/botkit

[7] "Spark architecture",
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-architecture.html

[8] "Spark stack",
https://mapr.com/ebooks/spark/03-apache-spark-architecture-overview.html

[9] "Benchmarking results by Databricks",
https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html

[10] "Spark streaming", http://spark.apache.org/docs/latest/streaming-programming-guide.html

[11] "Amazon Kinesis streams concepts",
http://docs.aws.amazon.com/streams/latest/dev/key-concepts.html

[12] "Amazon EC2 setup guide"
http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html

[13] "Deep learning in Natural Language Processing by Stanford",
https://nlp.stanford.edu/projects/DeepLearningInNaturalLanguageProcessing.shtml