Robust Feature Extraction Algorithm for Sarcasm Detection in Debates

CPSC 503 • December 2014

Olivia Norton & Issam H. Laradji

University of British Columbia

Abstract

Sarcasm is a subjective vocal inflection that is difficult for humans to consistently identify. The task of detecting sarcasm in text with no vocal cues is a complicated one. In this paper we propose a scheme for identifying whether a response to a statement in a debate is sarcastic. We explore a diverse set of textual features from low to high complexity to identify those that provide the most valuable information for sarcasm detection. The scheme is composed of four primary steps. First, we extract and select robust textual features. Second, we perform a novel unsupervised feature extraction using extreme learning machine (ELM) autoencoders which allow projection of a smaller set of features into a larger dimensional space. Third, feature selection is performed using chi-square and downsampling is done to account for the imbalance between sarcastic and non-sarcastic data. Finally, features are classified as either sarcastic or non-sarcastic using a simple logistic regression classifier. Autoencoded TF-IDF textual features are selected using chi-square and classified using simple logistic regression. We found a fair accuracy of 82.4% for detecting sarcasm using this configuration.

I. Introduction

Sarcasm in the field of NLP is a fairly young topic while humor detection and generation has been under the microscope for just a little longer (Mihalcea and Strapparava, 2006). There has been recent interest in using sarcasm in such applications as summarization, dialogue systems, and review ranking systems (Davidov et al., 2010). Most recently, the US secret service has released a work tender (BBC, 2014) describing a social media analytics tool for monitoring and visualizing various data. Specifically they have requested the ability to detect sarcasm, in addition to sentiment analysis and influencer identification. Previous approaches to humor detection stress the importance of ambiguity in creating the humorous effect (Reyes et al., 2010) (Krikmann, 2006). For sarcasm specifically, there have been attempts to use features such as the term ”yeah-right”, computing a measure of validity (vs. absurdity), as well as both syntactic and pattern-based features (Tsur et al., 2010), (Filatova, 2012). Taking advantage of changes wrought by social media, one group used the presence of hastags (#sarcasm) to generate a corpus of author annotates sarcastic Twitters comments.

In this paper we propose a system to identify sarcastic responses in a debate context. We present a diverse set of textual features of varying complexity with the intention of identifying those that provide the most valuable information for sarcasm detection. The algorithm first extracts and selects robust textual features. We then perform unsupervised feature extraction using extreme learning machine (ELM) autoencoders (ELM-AE) which allow projection of a smaller set of features into a larger dimensional space. Features are then classified as either sarcastic or non-sarcastic using a simple logistic regression classifier.

This paper presents three main contributions:

- A quantized evaluation of the importance of varying complexity features

- To apply unsupervised learning to extract more robust features from textual features
- To achieve state-of-the-art score on the sarcasm dataset.

I. Related Work

Automated recognition of sarcasm in text is a fairly novel task, though extensive study has previously covered the question of sarcasm in the context of linguistics (Utsumi, 1996).

Identification of sarcasm in a spoken dialogue system has been explored by (Tepperman et al., 2006). Their methods relied on analysis of the qualities of utterances including the statement ”yeah right”. In particular, the authors looked at characterizing the ”yeah right” as either acknowledgment, agreement/disagreement, indirect interpretation or internal to a phrase. This information in addition to a set of musical and spectral features is used to achieve strong results. The dataset used for this paper consisted of a set of recordings which included the phrase ”yeah right” Contextual features and spectral features combined provide the highest result with and accuracy of 87%.

News articles have been looked at by means of simple lexical features by (Burfoot and Baldwin, 2009). They were interested in identifying satirical vs. non-satirical articles. They focused their classification on three features types, headlines, the use of profanity, and the use of slang. They achieved an f-score of 0.798 on the corpus of news wire and satirical documents.

(Polanyi and Zaenen, 2006) proposed a proof of concept for improving results in negative vs. positive sentiment analysis. They utilize th[illegible]itudinal valence of lexical terms based on their context.

(Carvalho et al., 2009) investigated the ability to detect ironic sentences using surface patterns. They focus on positive predicates as they argue that these are more likely to contain irony. The authors perform classification on a corpus of Portuguese news articles and their associated comments. Particularly positive results, 85.4% accuracy, were reported when using ’laughter’ features. These features took advantage of internet slang for laughter including emoticons and acronyms like ’LOL’.

(Kreuz and Caucci, 2007) hypothesized that pragmatic and lexical factors have a part to play in the identification of sarcastic statements. In particular they focus on the presence of features related to adverbs/adjectives, punctuation and interjections. A corpus consisting of a collection of phrases selected using a Google Book Search for the term ’said sarcastically’ as well as ’said’, ’he said’ and ’she said’ was used for this work. A large group of participants then provided feedback as to whether they believed an utterance to be sarcastic. This information was combined with the annotations for adjectives, adverbs, interjections and use of punctuation. The results indicated that of the cues explored, only the presence of interjections proved significant in predicting sarcasm.

(González-Ibáñez et al., 2011) take advantage of the changing social media environment to create an author annotated corpus of sarcastic Twitter comments. They rely on the use of hashtags such as #sarcastic to identify and compile the corpus. They then use a set of lexical and pragmatic features to perform sarcastic utterance classification using logistic regressio[illegible]d support vector machine classifiers. The most positive results were based on the SVM classifier and basic unigram features with an average accuracy of 65.44[illegible]ova (Filatova, 2012) employed Mechanical Turk Workers to help in identifying and reviewing a corpus of sarcastic and non sarcastic product reviews on Amazon. The results of these experiments were a corpus annotated for sarcasm at both a sentence and document level.

(Tsur et al., 2010) and (Davidov et al., 2010) presented a semi-supervised sarcasm recognition system based on pattern recognition and punctuation and capital based features. Classification experiments were performed on two dataset, a Twitter dataset as well as a dataset composed of Amazon product reviews. The authors report a high accuracy of sarcastic utterance classification of 82.1%, though the recall and precision for that experiment were 31.2% and 25.6% respectively. The approach used by these authors to feature genera[illegible] is similar to the approach explored in this work.

II. Data

As is always the case when we begin exploration in a new direction of NLP, the need for annotated cor-

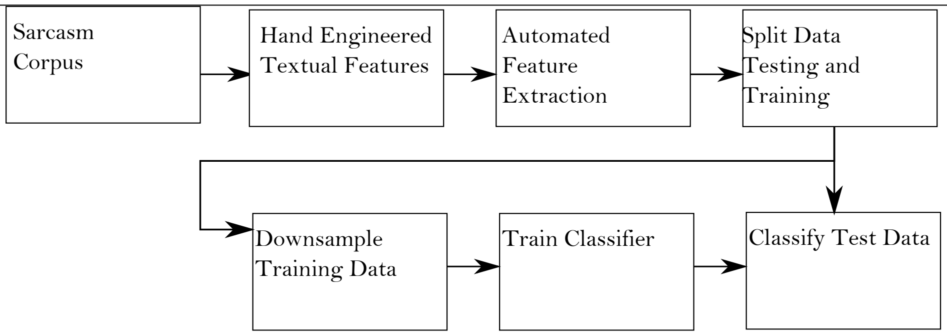
Sarcasm Corpus
Hand Engineered Textual Features
Automated Feature Extraction
Split Data Testing and Training
Downsample Training Data
Train Classifier
Classify Test Data

Figure 1: Sarcasm Recognition Framework

pora is of critical importance.

I. Corpus

A team out of the University of California Santa Cruz (Walker et al., 2012) has put together a corpus [1] of quote and response pairs scraped from online debate forums. It includes a set of 390,704 posts from 11,800 discussions focused on high controversy topics such as abortion, climate change, evolution, gun control and gay marriage, among others. The corpus was generated with the intention of facilitating research in the arena of deliberation and debate.
Corpus annotation was performed by Mechanical Turk for the following characteristics: Agree/Disagree, Fact/Emotion, Attack/Insult, Sarcasm,Nice/Nasty, Audience,Undercutting, Negotiate/Attack and Question/Assert. Each quote response pair was annotated by five separate Turkers. Analyses showed the task of analyzing for these criteria was difficult. To minimize noise, the authors used a two-level training and filtering framework to ensure only those Turkers who had a proven grasp of the language and the requirements were invited to annotate the final threads. It is important to note that previous work (Bryant and Fox Tree 2002) indicates that non-experts appear to group most forms of verbal irony into the single term of sarcasm, meaning that the system proposed below may, by proxy, also be capturing more than what might strictly be considered sarcasm. Table 1 reports some details about the corpus.

[1]Found here: https://nlds.soe.ucsc.edu/iac

#Sarcastic Samples	**#Non-Sarcastic Samples**	**# BoW Features**
1283	8694	33586

Table 1: Corpus information

III. High-level implementation

The scheme takes the following steps as shown in Figure 1.

1. Extract hand-engineered features
2. Extract latent features using ELM autoencoder
3. Apply feature selection and down-sampling to equalize the number of sarcastic samples with the number of non-sarcastic samples.
4. Use logistic regression to classify the samples into sarcastic and non-sarcastic.

These steps are explained in more detail in the following sections.

I. Extraction of Hand-Engineered Features

Verbal irony and sarcasm can be used in a multitude of ways from pointed commentary to subtle reproach. Depending on the context, the speaker or author may make it more or less obvious that the statement they are making is in fact sarcastic. Sarcasm, therefore, is a complicated affair. It generally requires a minimum level of universal or contextual knowledge to understand the nuances of a subtle statement, but other cues can and are often used including intonation in speech, as well as body language. As neither intonation, nor body language are easily read from text (with the exception of the convenient use of emoticons, emoticon_rolleyes), there must be other lexical, syntactic, and structural patterns which can aid in identification of text-based sarcasm. In this section we described the hand-engineered features explored in this paper. We select a subset of representative features of various types~~-~~. This will allow us to identify the types of patterns which have the most effect on a machine~~s~~ ability to detect sarcasm. In the feature section, we explored three main groups. The first is our baseline, described below. The second is an extended set of low and high complexity features which look at quote-response similarity among other things, and the third are robust TF-IDF features.

I.1 Baseline Features - BASE

1. **Sentence Length**
   The measure of sentence length in number of words provides a fairly simple way to assess the pointedness of the comment we are analyzing. As sarcasm generally manifests as a sharp or cutting comment, it follows that the shortness of the sentence may have some bearing on the conversation. Shortness of the sentence as well.
   This measure is used as part of our baseline features.

   Example
   Quote: Since the mass fatal shooting at Virginia Tech in 2007, gun-rights advocates have made an all-out effort to allow students to carry hidden firearms - on the dubious theory that students would be better protected from mass killers. But 22 states saw the folly of this idea and defeated it, even in strong gun-rights states such as Louisiana, South Carolina, and Oklahoma.
   Response: So students don't deserve their constitutional rights?

2. **Punctuation**
   Punctuation has been used as a marker for text based sarcasm in previous works (Davidov et al. 2010, Tsur et al. 2010, Carvalho et al. 2009). It has proven to be an important feature in these cases and so we include it as part of our classification as well. We focus on various normalize punctuation counts including question and exclamation marks, and quotations.
   Example
   Quote: And why is it that animals who don't often change their environments are the very same animals who havent "evolved" in millions of years...ie...crocodiles, sharks, bats, etc?

   Response: LOL. "Bats haven't evolved in millions of years"...

3. **Capitals**
   In text, we no longer ~~have the~~ use of vocal inflection to lend particular emphasis to certain words and phrases. It is common for authors the employ the creative and occasionally gratuitous use of capital letters to lend this vocal inflection to text based communications. In this section, two specific metrics are utilized. The first it the normalized number of capital letters in a response. The second is the normalized number of capitalized words in a sentence.

   Example
   Quote: If the christianists are dead set against we gay people getting married then I say lets let them keep marriage. Lets go on the attack and attempt to destroy christainist hetero marriages. We gay men need to suduce the men and the lesbians need to suduce the women. Lets see if we can drive those divorce rates up to 60%, 75%, or even 90%.

   Response: Oh, THANK YOU MATT! You just effectively shot down every argument I had in the 'indoctrinate our children' thread. See if I stick up for YOU publicly anymore.

I.2 Extended Features - EFEAT

1. **Punctuation**
   We extend the normalized punctuation counts described in the baseline to include commas and periods. The expectation is that these characters will also lend information to the presence or absence of sarcasm.
   Example
   Response: Obviously you have the answer to

this question, along with evidence that your answer is correct and factual, ...

2. **Word-Overlap**
   The word-overlap feature is a measure of not only similarity, but also of 'parroting'. Parroting in a response is the use of the exact structure, and word choices of a previous comment to make a pointed remark. This feature is computed as the normalized count of number of overlapping words between a quote and response.
   Example
   Quote: How about a sin tax of $100 each time you buy a gun and $10 each time you buy a bullet? Its fair because it would help pay for all the damage guns do to society. Rights come with responsibilities.

   Response: How about a sin tax of $100 dollars each time you log on and $10 dollars a word for each time you speak one?

3. **Similarity Score**
   The intuition leading to the use of this feature is base on the sarcastic utterances wherein the response either re-iterates the thoughts of the quote in a disbelieving way or completely changes the subject by associating the actions, or thoughts of their opponent to an event or situation that is completely unrelated. In the first case, we would expect to see fairly high similarity between quote and response, while the second case would lead to very dissimilar subject contents.
   We employed an LDA (Steyvers and Griffiths 2006)topic model over the entire corpus with the topic number empirically set to 40 topics. Euclidean distance between quote and response topics vectors was then computed and normalized.

   Example - Highly Similar
   Quote: A few Bible studies, comparing it to the flow of events, the nature of people, my shortcomings, the shortcomings of science etc, convinced me.

   Response: So your biased reading of the Bible coupled with your personal flaws and your misunderstanding of science is the basis of your religiosity? Somehow I don't think that's something you'd really want to brag about...

   Example - Highly Dis-similar
   Quote: Your reasoning of the effects of abortion are correct. The liberals who see abortion as a normal, acceptable, right to choose abortion are aborting themselves into extinction.

   Response: How tolerant. Lets talk about the Spanish Inquisition.

I.3 TF-IDF Features - TFIDF

For our scheme, we extract robust textual features known as Term Frequency Inverse Document (TF-IDF). For each quote or response, it first constructs a feature vector that represents the count of each term - like Bag of Words (Joachims, 2002). However, stop words such as 'a' and 'the' are ignored. Next, TF-IDF scales the term counts based on how many times the terms appear in the corpus. More formally, this is written as,

$$d_i = TF\left(w_i, doc_i\right) \cdot \log\left(\frac{|D|}{DF(w_i)}\right) \tag{1}$$

where doc_i [illegible] document i, d_i denotes the feature vector for doc_i, $TF(w_i, doc_i)$ is a function that returns how many times word (or term) w_i appears in doc_i, $|D|$ is the number of documents in the corpus, and $DF(w_i)$ returns the number of times w_i appears in the corpus. It is worth noting that the TF-IDF features have been commonly used for text representation (Ramos, 2003). This feature representation applies term weighting so that common words such as '*he*' and '*went*' are given less weight and more weight is assigned to less occurring words in the collection. This avoids common words from overshadowing the feature values represented by more distinguishing words.

Example
Quote: No they couldn't.
Response: Oh, OK then. emoticon_rolleyes

II. Feature Extraction using ELM-AE

Extreme learning machines (ELMs) have the ability to train very quickly yet develop a robust non-linear function (Huang et al., 2006). This makes it appropriate for natural language processing datasets that tend to be very large. Here we propose using an ELM network, known as ELM-autoencoder (ELM-AE), for feature extraction. It trains on the TF-IDF features and extracts new features that are retained in the hidden layer shown in Figure 2.

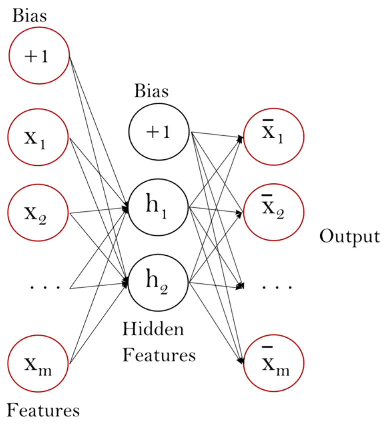

Figure 2: ELM-Autoencoder Network.

ELM-AE is described as follows. Given an input matrix $X \in \mathbb{R}^{n \times m}$, a bias vector $b \in \mathbb{R}^{L}$, and weight matrices $W \in \mathbb{R}^{m \times L}$ and $\beta \in \mathbb{R}^{L \times m}$. Consider a network containing m input neurons, L hidden neurons, and m output neurons. The function to optimize is written as,

$$L(\beta) = min_{\beta}\frac{1}{2}||\bar{X} - g(Xw + b)\beta||_2^2 + \frac{1}{2}\lambda||\beta||_2^2 \tag{2}$$

where $g(A) = \max(0, A)$ which is the Rectified Linear Unit Function (ReLU), and $\bar{X}$, the target value, is set as X. In other words, ELM-AE learns to find features $H = g(Xw + b)$ that can reconstruct the original features X.

Matrix w is uniformly randomized between a small range of values. Then, the hidden activations H of the hidden layer are computed as,

$$H = g(X \cdot W + b) \tag{3}$$

This gives us,

$$L(\beta) = min_{\beta}\frac{1}{2}||\bar{X} - H\beta||_2^2 + \frac{1}{2}\lambda||\beta||_2^2 \tag{4}$$

which can be solved using least squares. The goal is to find β that minimizes equation 4. Taking the derivative with respect to β for equation 4 and equating it to zero, we get,

$$\begin{aligned} H^T(H\beta - \bar{X}) + \lambda\beta \\ &= H^T H\beta - H^T\bar{X} + \lambda\beta \\ &= (H^T H + \lambda I)\beta - H^T\bar{X} = 0 \end{aligned} \tag{5}$$

β can then be solved with regularization as follows,

$$\beta = (\lambda I + H^T H)^{-1} H^T \bar{X} \tag{6}$$

where I is the identity matrix, and λ is a constant that controls the regularization term. Lower λ leads to learning a more linear function, as it increases bias and becomes less affected by variations (such as those caused by noisy data) in the dataset.

Finally, the new features F are computed using this equation,

$$F = g(X \cdot \beta^T) \tag{7}$$

These new features represent interesting structural information about the input TF-IDF features. Our results show that these features are robust enough to allow linear models such as logistic regression to perform as efficient as non-linear models such as extreme learning machine classifier.
Note that there is a tuneable parameter describing the number of hidden neurons used in the ELM autoencoder. A description of how this parameter is set is included in section IV.I Experimental Setup.

III. Feature Selection using Chi-Square

TF-IDF tends to generate high-dimensional sparse feature vectors for each response sentence which can hurt generalization (Sun et al., 2012). To ameliorate this, we use chi-square statistics to reduce the feature space and retain those features that are best correlated with the target value. The correlation between feature k and the labels is computed as follows,

$$\tilde{\chi_k}^2 = \frac{(O_k - E)^2}{E} \tag{8}$$

where,

$$O_k = y^T \cdot F_k \tag{9}$$

$$E_k = \frac{1}{n}\sum_{i=1}^{n} y_i \sum_{j=1}^{n} F_{kj} \tag{10}$$

where y is the label vector, F_k is the vector representing feature k, and $\sum_{j=1}^{n} F_{kj}$ is the summation over vector F_k. The features with the highest $\tilde{\chi_k}^2$ are retained.
The number of features to be selected is a tunable parameter that we explore. In section IV.I (Experimental Setup), we explain how we select that parameter.

IV. Logistic Regression Classification

Logistic regression is a fast, efficient classifier for training datasets with large number of features. This classifier minimizes the error that is assumed to fall under the logistic function as depicted in eq. (6) and Figure 3,

$$Pr(Y_i = y_i|X_i) = \frac{1}{1 + \exp^{-wX_i}} \tag{11}$$

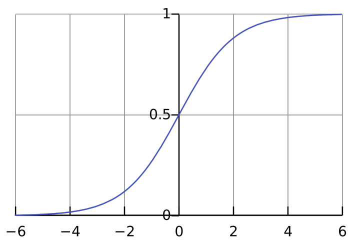

Figure 3: Logistic Curve

This generates a linear model for classification. Since logistic regression requires a balanced ratio between classes to preform efficiently, we downsampled the training set by randomly removing samples falling under the non-sarcastic label. With such smaller training set and equal ratio of class sizes, logistic regression achieved state-of-the-art results for our scheme.

IV. Experimentation

I. Experimental Setup

Experiments were run on a machine with 3.6 GHz quad-core CPU and 8 GB RAM operating a 64-bit Windows 8. We evaluate our schemes through a 10-fold stratified cross-validation method where the dataset is divided into 90% training and 10% testing for each fold. As such, both sets have similar ratio of positive samples to negative samples. The scores are based on average accuracy, recall and precision. In other words, the scores for the sarcastic class are computed alone, and for the non-sarcastic class are computed alone as well. The final score is the average between them. The reason for averaging the scores between classes is because the dataset is imbalanced as non-sarcastic samples highly outnumber sarcastic samples.

The recall measure defines the ratio of the number of correctly classified documents in the category to the total number of documents in the category:

$$Recall = \frac{TP}{TP + FN} \tag{12}$$

The precision is the ratio of correctly classified documents in the category to the total number of documents classified in the category:

$$Precision = \frac{TP}{TP + FP} \tag{13}$$

The accuracy is calculated as,

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{14}$$

For the benchmark, we evaluated the following schemes,

1. Scheme 1: TFIDF + ELM-AE + LOG
TF-IDF feature extraction is performed. Features undergo ELM auto-encoding to identify latent structures. Feature selection is performed using chi-square and downsampling is applied to the data. Finally classification is performed using a simple logistic regression classifier.

2. Scheme 2: TFIDF + LOG
TF-IDF feature extraction is performed. No feature autoencoding is performed. Feature selection is performed using chi-square and downsampling is applied to the data. Finally classification is performed using a simple logistic regression classifier.

3. Scheme 3: BASE + KNN
Baseline features per section 3.1.1.1 are extracted. No feature autoencoding or chi-square

selection is performed. Data is down-sampled and classified using a 5-Nearest-Neighbor classifier.

4. Scheme 4: BASE + LOG
   Baseline features per section 3.1.1.1 are extracted. No feature autoencoding or chi-square selection is performed. Data is down-sampled and classified using a logistic regression classifier.

5. Scheme 5: BASE + EFEAT + TFIDF + LOG
   All features including baseline (3.1.1.1), extended (3.1.1.2) and TF-IDF (3.1.1.3) features are extracted then undergo chi-square selection and down-sampling. Data is then classified using a logistic regression classifier.

6. Scheme 6: BASE + EFEAT + TFIDF + KNN
   All features including baseline (3.1.1.1), extended (3.1.1.2) and TF-IDF (3.1.1.3) features are extracted then undergo chi-square selection and down-sampling. Data is then classified using a logistic regression classifier.

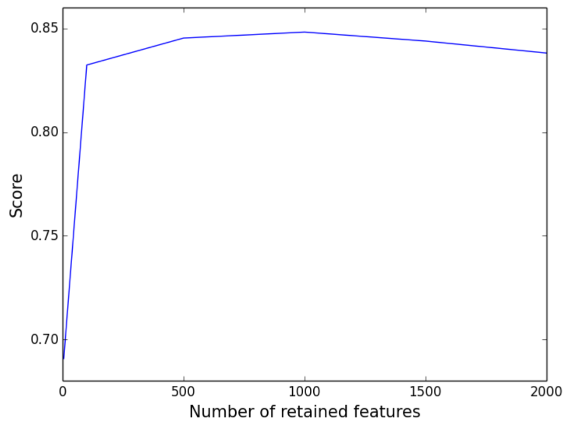

Figure 4: Effects of Varying Number Chi-Square Selected Features on Accuracy.

There are two tuneable parameters in this algorithm. In order to optimize algorithm results, simple experiments varying the number of ELM hiddden neurons and chi-square features ~~was~~ performed using a validation set that constitutes 20% of the training set. Figures 4 and 5 display the results. The number of hidden neurons is therefore empirically set to 6000, and the number of retained chi-square features was set to 1500.

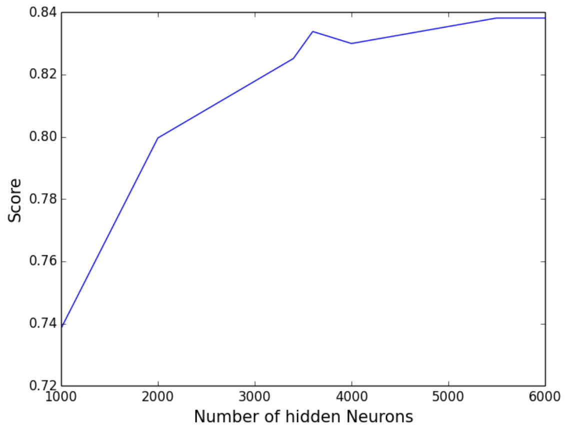

Figure 5: Effects of Varying Number ELM Hidden Neurons

#	Scheme	Acc.	Rec.	Prec.
1	**TFIDF+ELM+LOG**	**0.824**	**0.819**	**0.824**
2	TFIDF+LOG	0.676	0.676	0.676
3	BASE+KNN	0.524	0.499	0.498
4	BASE+LOG	0.543	0.539	0.543
5	BASE+EFEAT+LOG	0.717	0.693	0.696
6	BASE+EFEAT+KNN	0.537	0.504	0.537

Table 2: Experimental Results

V. Evaluation and Results

We performed several tests varying the combinations of simple manually extracted features, and classification methods to identify the particular combinations which provided the best results. We chose a total of 6 evaluation schemes as described in the previous section. Table 2 shows the results of these tests. Figure 6 displays the accuracies of scheme 1 and 2 with respect to the number of features selected by the chi-squared method. We see stabilization of results around the 30 feature mark. The confusion matrices of the two schemes are given in Table 4 and Table 3, respectively. The combination of simple TF-IDF features with an additional ELM-AE feature extraction step and linear regression classification resulted in the best outcome with an average accuracy of 82.4

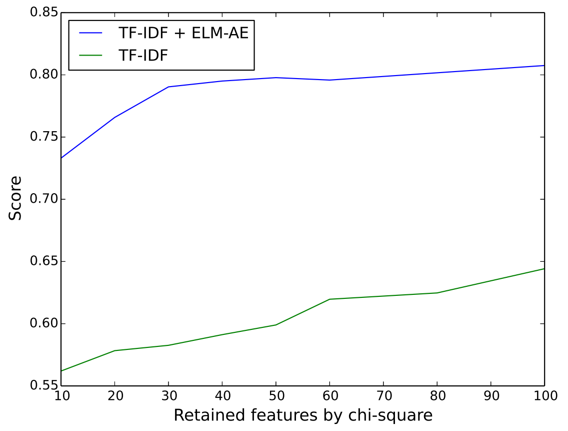

Figure 6: Accuracy of TF-IDF feature extraction with and without ELM autoencoding.

	T-S	T-NS
C-S	90	38
C-NS	44	83

Table 3: Confusion matrix representing results of testing scheme 2. TF-IDF Features with no ELM autoencoder. T-S and T-NS are the true Sarcastic and non-sarcastic labels respectively, and C-S and C-NS show the results of our classification.

VI. Discussion

We note fairly comparable results between schemes 2 and 5 which nicely demonstrates the competition between specific hand-engineered features, and holistic statistical features. The ELM-AE step was not used to project a subset of features into a larger dimensional space for scheme 5 primarily due to the limited number of features in this experiment. The ELM-AE is particularly valuable when we have a large subset of features to begin with. As we know that sarcasm components can be extremely complicated, the superior performance of the lower level auto-encoded TF-IDF features is not intuitive. TF-IDF has been shown to be very capable of text representation in other applications, and it has demonstrated that ability again here.

	T-S	T-NS
C-S	111	17
C-NS	31	98

Table 4: Confusion matrix representing results of testing scheme 1. TF-IDF Features with ELM autoencoder. T-S and T-NS are the true Sarcastic and non-sarcastic labels respectively, and C-S and C-NS show the results of our classification.

The use of ELM autoencoding shows a significant improvement in accuracy for extracted TF-IDF features. ELM is able to discern robust structural features in the data where input features are seemingly correlated. The KNN classifier performs very poorly. This is not unexpected as KNN is known to have a high variance and therefore weak results in a high dimensional classification space (Weber et al., 1998).

VII. Conclusion

A novel approach to classification of sarcastic statements based on simple automatically extracted features has been presented. Very positive fair accuracy measure of 82.4 have been reported, providing concrete support for the ongoing use of simple features. Extreme Learning machines have shown themselves to be extremely valuable intermittent step allowing the projection of a subset of features into a higher dimensional space. These features are robust enough to allow linear models to develop an efficient classification decision boundary for sarcasm detection.

I. Future Work

It would be very interesting to continue to look at the results of extending the list of manually engineered textual features. Two areas of specific interest are the use of Part-Of-Speech information as well as syntactic sentence structure to aid in the classification of sarcasm. These types of features would obviously require increased computational overhead, and may prove themselves to be more expensive than they are worthwhile. The debate corpus used in this implementation certainly had good examples of sarcasm for training and classification, but, as always, we wish for a domain independent solution to sarcasm detection. It would be of value to apply the existing algorithm to an extend set of corpora, and identify strengths and domain-dependent weaknesses in the current approach. The TF-IDF weighting scheme could be improved upon by exploring class specific weighting. This type of approach would take advantage of words which occur more commonly in one specific class of another.

References

BBC. 2014. Us secret service seeks twitter sarcasm detector. In *http://www.bbc.com/news/technology-27711109*. BBC News.

Clint Burfoot and Timothy Baldwin. 2009. Automatic satire detection: Are you having a laugh? In *Proceedings of the ACL-IJCNLP 2009 conference short papers*, pages 161–164. Association for Computational Linguistics.

Paula Carvalho, Luís Sarmento, Mário J Silva, and Eugénio de Oliveira. 2009. Clues for detecting irony in user-generated contents: oh...!! it's so easy;-). In *Proceedings of the 1st international CIKM workshop on Topic-sentiment analysis for mass opinion*, pages 53–56. ACM.

Dmitry Davidov, Oren Tsur, and Ari Rappoport. 2010. Semi-supervised recognition of sarcastic sentences in twitter and amazon. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning*, pages 107–116. Association for Computational Linguistics.

Elena Filatova. 2012. Irony and sarcasm: Corpus generation and analysis using crowdsourcing. In *LREC*, pages 392–398.

Roberto González-Ibáñez, Smaranda Muresan, and Nina Wacholder. 2011. Identifying sarcasm in twitter: a closer look. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 581–586. Association for Computational Linguistics.

Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. 2006. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1):489–501.

Thorsten Joachims. 2002. *Learning to classify text using support vector machines: Methods, theory and algorithms*. Kluwer Academic Publishers.

Roger J Kreuz and Gina M Caucci. 2007. Lexical influences on the perception of sarcasm. In *Proceedings of the Workshop on computational approaches to Figurative Language*, pages 1–4. Association for Computational Linguistics.

Arvo Krikmann. 2006. Contemporary linguistic theories of humour. *Folklore: Electronic Journal of Folklore*, (33):27–58.

Rada Mihalcea and Carlo Strapparava. 2006. Technologies that make you smile: Adding humor to text-based applications. *Intelligent Systems, IEEE*, 21(5):33–39.

Livia Polanyi and Annie Zaenen. 2006. Contextual valence shifters. In *Computing attitude and affect in text: Theory and applications*, pages 1–10. Springer.

Juan Ramos. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the First Instructional Conference on Machine Learning*.

Antonio Reyes, Davide Buscaldi, and Paolo Rosso. 2010. The impact of semantic and morphosyntactic ambiguity on automatic humour recognition. In *Natural language processing and information systems*, pages 130–141. Springer.

Zhongbin Sun, Qinbao Song, and Xiaoyan Zhu. 2012. Using coding-based ensemble learning to improve software defect prediction. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(6):1806–1817.

Joseph Tepperman, David R Traum, and Shrikanth Narayanan. 2006. " yeah right": sarcasm recognition for spoken dialogue systems. In *INTERSPEECH*.

Oren Tsur, Dmitry Davidov, and Ari Rappoport. 2010. Icwsm-a great catchy name: Semi-supervised recognition of sarcastic sentences in online product reviews. In *ICWSM*.

Akira Utsumi. 1996. A unified theory of irony and its computational formalization. In *Proceedings of the 16th conference on Computational linguistics-Volume 2*, pages 962–967. Association for Computational Linguistics.

Marilyn A Walker, Jean E Fox Tree, Pranav Anand, Rob Abbott, and Joseph King. 2012. A corpus for research on deliberation and debate. In *LREC*, pages 812–817.

Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205.

VIII. Appendix

The corpus is found here:
https://nlds.soe.ucsc.edu/iac
The source code is in the online copy.

I. Demo File

```
import nltk

import pandas as pd
import numpy as np
import sklearn.neural_network
import utilities as ut
import utilities_olivia as olivia

from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import normalize
from sklearn.svm import SVC
from extreme_learning_machines import ELMRegressor, ELMClassifier
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.metrics import confusion_matrix
from sklearn.ensemble import RandomForestClassifier
from sklearn.cross_validation import train_test_split
from sklearn.feature_selection import chi2
from sklearn.feature_selection import SelectKBest
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score
from sklearn import cross_validation
from sklearn import preprocessing
from scipy import sparse
from sklearn.neighbors import KNeighborsClassifier
from sklearn import cross_validation
from sklearn.metrics import fbeta_score, make_scorer
from sklearn.cross_validation import StratifiedShuffleSplit

scorer = make_scorer(ut.fair_accuracy)

def plot_wrt_chisquare(X, y, chi_square_list=[2000], n_hidden_list=[6000], \
                       with_feature_extraction=False):

    if len(chi_square_list) == 1:
        score_list = np.zeros(len(n_hidden_list))
    else:
        score_list = np.zeros(len(chi_square_list))

    for i, chi in enumerate(chi_square_list):
        for j, n_hidden in enumerate(n_hidden_list):
            X_, y_ = ut.extract_ELM_features(X, y, with_feature_extraction,
                                             chi_square = chi,
                                             n_hidden=n_hidden)
            clf = LogisticRegression()
            score = np.mean(cross_validation.cross_val_score(clf, X_, y_, cv=10,\
                                  scoring = scorer))

            if len(chi_square_list) == 1:
                    score_list[j] = score
            else:
                    score_list[i] = score


    print score_list


def compute_score(X, y, chi_square=1500, n_hidden=6000, \
                  with_feature_extraction=False):
    X_, y_ = ut.extract_ELM_features(X, y, with_feature_extraction,
                                     chi_square = chi_square,
                                     n_hidden=n_hidden)
    clf = LogisticRegression()
    score = np.mean(cross_validation.cross_val_score(clf, X_, y_, cv=10,
                                                      scoring = scorer))


# Read the 'qr_meta.csv' excel sheet containing the quote-response pairs
location = "D:/datasets/iac_v1.1/data/fourforums/annotations/" \
         "mechanical_turk/"



chi_square_list = [5, 100, 500, 1000, 1500, 2000]
X, y = ut.read_dataset(location)
X, y = X[:8000], y[:8000]
y = y.flatten()
transformer = TfidfVectorizer(stop_words="english", sublinear_tf =True)
X = transformer.fit_transform(X)
plot_wrt_chisquare(X, y, chi_square_list=chi_square_list, with_feature_extraction =
wqewqe

### Results for TFIDF+ELM
chi_square_list = [10, 20, 30, 40, 50, 60, 80, 100]
X, y = ut.read_dataset(location)
y = y.flatten()
transformer = TfidfVectorizer(stop_words="english", sublinear_tf =True)
X = transformer.fit_transform(X)
#plot_wrt_chisquare(X, y, chi_square_list, with_feature_extraction = True)


#sadas
### Results for TFIDF
chi_square_list = [10, 20, 30, 40, 50, 60, 80, 100]
```

```
X, y = ut.read_dataset(location)
y = y.flatten()
transformer = TfidfVectorizer(stop_words="english", sublinear_tf =True)
X = transformer.fit_transform(X)
#plot_wrt_chisquare(X, y, chi_square_list, with_feature_extraction = False)

### Results for TFIDF+ELM
n_hidden_list = [3000, 3400, 3600, 4000, 5500]
X, y = ut.read_dataset(location)
y = y.flatten()
transformer = TfidfVectorizer(stop_words="english", sublinear_tf =True)
X = transformer.fit_transform(X)
#plot_wrt_chisquare(X, y, n_hidden_list=n_hidden_list, with_feature_extraction = T

### Results for BoW+ELM
n_hidden_list = [2000, 2400, 2600, 3000, 4500]
X, y = ut.read_dataset(location)
y = y.flatten()
transformer = CountVectorizer(stop_words="english")
X = normalize(transformer.fit_transform(X).todense())
#plot_wrt_chisquare(X, y, n_hidden_list=n_hidden_list, with_feature_extraction = F
################ Accuracy, recall, precision


# TF-IDF + ELM (Accuracy : 0.824)
X, y = ut.read_dataset(location)
y = y.flatten()
transformer = TfidfVectorizer(stop_words="english", sublinear_tf =True)
X = transformer.fit_transform(X)
print "TF-IDF + ELM"
compute_score(X, y, with_feature_extraction=True)

# TF-IDF (Accuracy : 0.676)
X, y = ut.read_dataset(location)
y = y.flatten()
transformer = TfidfVectorizer(stop_words="english", sublinear_tf =True)
X = transformer.fit_transform(X)
print "TF-IDF"
compute_score(X, y, with_feature_extraction=False)

# baseline + KNN (Accuracy : 0.524)
dataset, y = olivia.read_dataset_whole(location)
X = olivia.getBaselineFeatures(dataset)
y = y.flatten()

clf = KNeighborsClassifier(n_neighbors=5, weights='distance')
score = np.mean(cross_validation.cross_val_score(clf, X, y, cv=3, \
                scoring = scorer))
```

```
print "baseline + KNN"
print score

# baseline + logistic (0.543)
dataset, y = olivia.read_dataset_whole(location)
X = olivia.getBaselineFeatures(dataset)
y = y.flatten()
X, y = ut.balance_dataset(X, y)
clf = LogisticRegression()
score = np.mean(cross_validation.cross_val_score(clf, X, y, cv=3, \
                scoring = scorer))
print "baseline + logistic"
print score
# baseline + similarity features + logistic (Accuracy: 0.71659)
X = np.load("ldafeatures_X.npy")
y = np.load("ldafeatures_y.npy")
y=y.flatten()
X, y = ut.balance_dataset(X, y)
clf = LogisticRegression()
score = np.mean(cross_validation.cross_val_score(clf, X, y, cv=3, \
                 scoring = scorer))

print "baseline + similarity features + logistic"
print score

# baseline + TF-IDF + similarity features + KNN (Accuracy: 0.537)
X = np.load("ldafeatures_X.npy")
y = np.load("ldafeatures_y.npy")
y=y.flatten()
clf = KNeighborsClassifier(n_neighbors=5, weights='distance')
score = np.mean(cross_validation.cross_val_score(clf, X, y, cv=3,\
                 scoring = scorer))
print "baseline + TF-IDF + similarity features + KNN"
print score


# baseline + similarity features + logistic (Accuracy: 0.71659)
X = np.load("ldafeatures_X.npy")
y = np.load("ldafeatures_y.npy")
y=y.flatten()
#X, y = ut.balance_dataset(X, y)
#compute_score(X, y, n_hidden=100, with_feature_extraction=True)

#print X

compute_score(X, y, with_feature_extraction=True)
#print score
```

```
X, y = ut.read_dataset(location)
y = y.flatten()
transformer = TfidfVectorizer(stop_words="english", sublinear_tf =True)
X = transformer.fit_transform(X)
```

II. ELM-AE File

```
"""Extreme␣Learning␣Machines
"""

# Author: Issam H. Laradji <issam.laradji@gmail.com>
# Licence: BSD 3 clause


from abc import ABCMeta, abstractmethod

import numpy as np
from scipy import linalg

from sklearn.base import BaseEstimator, ClassifierMixin, RegressorMixin
from base import logistic, softmax, ACTIVATIONS
from sklearn.externals import six
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import mean_squared_error
from sklearn.linear_model.ridge import ridge_regression
from sklearn.utils import gen_batches, check_random_state
from sklearn.utils.extmath import safe_sparse_dot
#from ..utils import check_array, check_X_y, column_or_1d
from class_weight import compute_sample_weight
from sklearn.utils import atleast2d_or_csr, check_arrays


def _multiply_weights(X, sample_weight):
    """Return␣W*X␣if␣sample_weight␣is␣not␣None."""
    if sample_weight is None:
        return X
    else:
        return X * sample_weight[:, np.newaxis]


class BaseELM(six.with_metaclass(ABCMeta, BaseEstimator)):
    """Base␣class␣for␣ELM␣classification␣and␣regression.

␣␣␣␣Warning:␣This␣class␣should␣not␣be␣used␣directly.
␣␣␣␣Use␣derived␣classes␣instead.
␣␣␣␣"""
    @abstractmethod
    def __init__(self, n_hidden, activation, C, class_weight,
                 weight_scale, batch_size, verbose, warm_start,
                 random_state):
        self.C = C
        self.activation = activation
        self.class_weight = class_weight
        self.weight_scale = weight_scale
        self.batch_size = batch_size
        self.n_hidden = n_hidden
        self.verbose = verbose
        self.warm_start = warm_start
        self.random_state = random_state

    def _init_weights(self, n_features):
        """Initialize the parameter weights."""
        rng = check_random_state(self.random_state)

        # Use the initialization method recommended by Glorot et al.
        weight_init_bound = np.sqrt(6. / (n_features + self.n_hidden))

        self.coef_hidden_ = rng.uniform(-weight_init_bound,
                                        weight_init_bound, (n_features,
                                                            self.n_hidden))
        self.intercept_hidden_ = rng.uniform(-weight_init_bound,
                                             weight_init_bound,
                                             self.n_hidden)
        if self.weight_scale != 1:
            self.coef_hidden_ *= self.weight_scale
            self.intercept_hidden_ *= self.weight_scale

    def _compute_hidden_activations(self, X):
        """Compute the hidden activations."""

        hidden_activations = safe_sparse_dot(X, self.coef_hidden_)
        hidden_activations += self.intercept_hidden_

        # Apply the activation method
        activation = ACTIVATIONS[self.activation]
        hidden_activations = activation(hidden_activations)

        return hidden_activations

    def _fit(self, X, y, sample_weight=None, incremental=False):
        """Fit the model to the data X and target y."""
        # Validate input params
        if self.n_hidden <= 0:
            raise ValueError("n_hidden must be > 0, got %s." % self.n_hidden)
        if self.C <= 0.0:
            raise ValueError("C must be > 0, got %s." % self.C)
        if self.activation not in ACTIVATIONS:
            raise ValueError("The activation %s is not supported. Supported "
                             "activation are %s." % (self.activation,
                                                      ACTIVATIONS))

        # Initialize public attributes
        if not hasattr(self, 'classes_'):
            self.classes_ = None
        if not hasattr(self, 'coef_hidden_'):
            self.coef_hidden_ = None

        # Initialize private attributes
        if not hasattr(self, '_HT_H_accumulated'):
            self._HT_H_accumulated = None

        X, y = check_arrays(X, y)

        # This outputs a warning when a 1d array is expected
        #if y.ndim == 2 and y.shape[1] == 1:
        #    y = column_or_1d(y, warn=True)

        # Classification
        if isinstance(self, ClassifierMixin):
            self.label_binarizer_.fit(y)

            if self.classes_ is None or not incremental:
                self.classes_ = self.label_binarizer_.classes_
                if sample_weight is None:
                    sample_weight = compute_sample_weight(self.class_weight,
                                                          self.classes_, y)
            else:
                classes = self.label_binarizer_.classes_
                if not np.all(np.in1d(classes, self.classes_)):
                    raise ValueError("`y` has classes not in `self.classes_`."
                                     " `self.classes_` has %s. 'y' has %s." %
                                     (self.classes_, classes))

            y = self.label_binarizer_.transform(y)

        # Ensure y is 2D
        if y.ndim == 1:
            y = np.reshape(y, (-1, 1))

        n_samples, n_features = X.shape
        self.n_outputs_ = y.shape[1]

        # Step (1/2): Compute the hidden layer coefficients
        if (self.coef_hidden_ is None or (not incremental and
                                          not self.warm_start)):
```

```
        # Randomize and scale the input-to-hidden coefficients
        self._init_weights(n_features)

    # Step (2/2): Compute hidden-to-output coefficients
    if self.batch_size is None:
        # Run the least-square algorithm on the whole dataset
        batch_size = n_samples
    else:
        # Run the recursive least-square algorithm on mini-batches
        batch_size = self.batch_size

    batches = gen_batches(n_samples, batch_size)

    # (First time call) Run the least-square algorithm on batch 0
    if not incremental or self._HT_H_accumulated is None:
        batch_slice = next(batches)
        H_batch = self._compute_hidden_activations(X[batch_slice])

        # Get sample weights for the batch
        if sample_weight is None:
            sw = None
        else:
            sw = sample_weight[batch_slice]

        # beta_{0} = inv(H_{0}^T H_{0} + (1. / C) * I) * H_{0}.T y_{0}
        self.coef_output_ = ridge_regression(H_batch, y[batch_slice],
                                             1. / self.C,
                                             sample_weight=sw).T

        # Initialize K if this is batch based or partial_fit
        if self.batch_size is not None or incremental:
            # K_{0} = H_{0}^T * W * H_{0}
            weighted_H_batch = _multiply_weights(H_batch, sw)
            self._HT_H_accumulated = safe_sparse_dot(H_batch.T,
                                                     weighted_H_batch)

        if self.verbose:
            y_scores = self._decision_scores(X[batch_slice])

            if self.batch_size is None:
                verbose_string = "Training mean squared error ="
            else:
                verbose_string = "Batch 0, Training mean squared error ="

            print("%s %f" % (verbose_string,
                             mean_squared_error(y[batch_slice], y_scores,
                                                sample_weight=sw)))
    # Run the least-square algorithm on batch 1, 2, ..., n
    for batch, batch_slice in enumerate(batches):
        # Compute hidden activations H_{i} for batch i
        H_batch = self._compute_hidden_activations(X[batch_slice])

        # Get sample weights (sw) for the batch
        if sample_weight is None:
            sw = None
        else:
            sw = sample_weight[batch_slice]

        weighted_H_batch = _multiply_weights(H_batch, sw)

        # Update K_{i+1} by H_{i}^T * W * H_{i}
        self._HT_H_accumulated += safe_sparse_dot(H_batch.T,
                                                   weighted_H_batch)

        # Update beta_{i+1} by
        # K_{i+1}^{-1} * H_{i+1}^T * W * (y_{i+1} - H_{i+1} * beta_{i})
        y_batch = y[batch_slice] - safe_sparse_dot(H_batch,
                                                    self.coef_output_)

        weighted_y_batch = _multiply_weights(y_batch, sw)
        Hy_batch = safe_sparse_dot(H_batch.T, weighted_y_batch)

        # Update hidden-to-output coefficients
        regularized_HT_H = self._HT_H_accumulated.copy()
        regularized_HT_H.flat[::self.n_hidden + 1] += 1. / self.C

        # It is safe to use linalg.solve (instead of linalg.lstsq
        # which is slow) since it is highly unlikely that
        # regularized_HT_H is singular due to the random
        # projection of the first layer and 'C' regularization being
        # not dangerously large.
        self.coef_output_ += linalg.solve(regularized_HT_H, Hy_batch,
                                          sym_pos=True, overwrite_a=True,
                                          overwrite_b=True)
        if self.verbose:
            y_scores = self._decision_scores(X[batch_slice])
            print("Batch %d, Training mean squared error = %f" %
                  (batch + 1, mean_squared_error(y[batch_slice], y_scores,
                                                 sample_weight=sw)))
    return self

def fit(self, X, y, sample_weight=None):
    """Fit the model to the data X and target y.

    Parameters
```

```
␣␣␣␣␣␣␣␣----------
␣␣␣␣␣␣␣␣X␣:␣{array-like,␣sparse␣matrix},␣shape␣(n_samples,␣n_features)
␣␣␣␣␣␣␣␣␣␣␣␣The␣input␣data.

␣␣␣␣␣␣␣␣y␣:␣array-like,␣shape␣(n_samples,)
␣␣␣␣␣␣␣␣␣␣␣␣Target␣values.

␣␣␣␣␣␣␣␣sample_weight␣:␣array-like,␣shape␣(n_samples,)
␣␣␣␣␣␣␣␣␣␣␣␣Per-sample␣weights.␣Rescale␣C␣per␣sample.␣Higher␣weights
␣␣␣␣␣␣␣␣␣␣␣␣force␣the␣classifier␣to␣put␣more␣emphasis␣on␣these␣points.

␣␣␣␣␣␣␣␣Returns
␣␣␣␣␣␣␣␣-------
␣␣␣␣␣␣␣␣self␣:␣returns␣a␣trained␣ELM␣usable␣for␣prediction.
␣␣␣␣␣␣␣␣"""
        return self._fit(X, y, sample_weight=sample_weight, incremental=False)

    def partial_fit(self, X, y, sample_weight=None):
        """Fit␣the␣model␣to␣the␣data␣X␣and␣target␣y.

␣␣␣␣␣␣␣␣Parameters
␣␣␣␣␣␣␣␣----------
␣␣␣␣␣␣␣␣X␣:␣{array-like,␣sparse␣matrix},␣shape␣(n_samples,␣n_features)
␣␣␣␣␣␣␣␣␣␣␣␣Subset␣of␣training␣data.

␣␣␣␣␣␣␣␣y␣:␣array-like,␣shape␣(n_samples,)
␣␣␣␣␣␣␣␣␣␣␣␣Subset␣of␣target␣values.

␣␣␣␣␣␣␣␣sample_weight␣:␣array-like,␣shape␣(n_samples,)
␣␣␣␣␣␣␣␣␣␣␣␣Per-sample␣weights.␣Rescale␣C␣per␣sample.␣Higher␣weights
␣␣␣␣␣␣␣␣␣␣␣␣force␣the␣classifier␣to␣put␣more␣emphasis␣on␣these␣points.

␣␣␣␣␣␣␣␣Returns
␣␣␣␣␣␣␣␣-------
␣␣␣␣␣␣␣␣self␣:␣returns␣a␣trained␣ELM␣usable␣for␣prediction.
␣␣␣␣␣␣␣␣"""
        self._fit(X, y, sample_weight=sample_weight, incremental=True)

        return self

    def _decision_scores(self, X):
        """Predict␣using␣the␣ELM␣model

␣␣␣␣␣␣␣␣Parameters
␣␣␣␣␣␣␣␣----------
␣␣␣␣␣␣␣␣X␣:␣{array-like,␣sparse␣matrix},␣shape␣(n_samples,␣n_features)
␣␣␣␣␣␣␣␣␣␣␣␣The␣input␣data.
```

```
        Returns
        -------
        y_pred : array-like, shape (n_samples,) or (n_samples, n_outputs)
            The predicted values.
        """
        #X = check_array(X, accept_sparse=['csr', 'csc', 'coo'])

        if self.batch_size is None:
            hidden_activations = self._compute_hidden_activations(X)
            y_pred = safe_sparse_dot(hidden_activations, self.coef_output_)
        else:
            n_samples = X.shape[0]
            batches = gen_batches(n_samples, self.batch_size)

            y_pred = np.zeros((n_samples, self.n_outputs_))
            for batch in batches:
                h_batch = self._compute_hidden_activations(X[batch])
                y_pred[batch] = safe_sparse_dot(h_batch, self.coef_output_)

        return y_pred


class ELMClassifier(BaseELM, ClassifierMixin):
    """Extreme learning machine classifier.

    The algorithm trains a single-hidden layer feedforward network by computing
    the hidden layer values using randomized parameters, then solving
    for the output weights using least-square solutions. For prediction,
    after computing the forward pass, the continuous output values pass
    through a gate function converting them to integers that represent classes.

    This implementation works with data represented as dense and sparse numpy
    arrays of floating point values for the features.

    Parameters
    ----------
    C : float, optional, default 100
        A regularization term that controls the linearity of the decision
        function. Smaller value of C makes the decision boundary more linear.

    class_weight : {dict, 'auto', None}, default None
        If 'auto', class weights will be given inversely proportional
        to the frequency of the class in the data.
        If a dictionary is given, keys are the class labels and the
        corresponding values are the class weights.
        If None is given, then no class weights will be applied.

    weight_scale : float, default 1.
```

```
␣␣␣␣␣␣␣␣Initializes␣and␣scales␣the␣input-to-hidden␣weights.
␣␣␣␣␣␣␣␣The␣weight␣values␣will␣range␣between␣plus␣and␣minus
␣␣␣␣␣␣␣␣'sqrt(weight_scale␣*␣6.␣/␣(n_features␣+␣n_hidden))'␣based␣on␣the
␣␣␣␣␣␣␣␣uniform␣distribution.

␣␣␣␣n_hidden␣:␣int,␣default␣100
␣␣␣␣␣␣␣␣The␣number␣of␣units␣in␣the␣hidden␣layer.

␣␣␣␣activation␣:␣{'logistic',␣'tanh',␣'relu'},␣default␣'relu'
␣␣␣␣␣␣␣␣Activation␣function␣for␣the␣hidden␣layer.

␣␣␣␣␣␣␣␣␣-␣'logistic',␣the␣logistic␣sigmoid␣function,
␣␣␣␣␣␣␣␣␣␣␣␣returns␣f(x)␣=␣1␣/␣(1␣+␣exp(x)).

␣␣␣␣␣␣␣␣␣-␣'tanh',␣the␣hyperbolic␣tan␣function,
␣␣␣␣␣␣␣␣␣␣␣␣returns␣f(x)␣=␣tanh(x).

␣␣␣␣␣␣␣␣␣-␣'relu',␣the␣rectified␣linear␣unit␣function,
␣␣␣␣␣␣␣␣␣␣␣␣returns␣f(x)␣=␣max(0,␣x).

␣␣␣␣batch_size␣:␣int,␣optional,␣default␣None
␣␣␣␣␣␣␣␣If␣None␣is␣given,␣batch_size␣is␣set␣as␣the␣number␣of␣samples.
␣␣␣␣␣␣␣␣Otherwise,␣it␣will␣be␣set␣as␣the␣given␣integer.

␣␣␣␣verbose␣:␣bool,␣optional,␣default␣False
␣␣␣␣␣␣␣␣Whether␣to␣print␣the␣training␣score.

␣␣␣␣warm_start␣:␣bool,␣optional,␣default␣False
␣␣␣␣␣␣␣␣When␣set␣to␣True,␣reuse␣the␣solution␣of␣the␣previous
␣␣␣␣␣␣␣␣call␣to␣fit␣as␣initialization,␣otherwise,␣just␣erase␣the
␣␣␣␣␣␣␣␣previous␣solution.

␣␣␣␣random_state␣:␣int␣or␣RandomState,␣optional,␣default␣None
␣␣␣␣␣␣␣␣State␣of␣or␣seed␣for␣random␣number␣generator.

␣␣␣␣Attributes
␣␣␣␣----------
␣␣␣␣`classes_`␣:␣array-list,␣shape␣(n_classes,)
␣␣␣␣␣␣␣␣Class␣labels␣for␣each␣output.

␣␣␣␣`n_outputs_`␣:␣int
␣␣␣␣␣␣␣␣Number␣of␣output␣neurons.

␣␣␣␣`coef_hidden_`␣:␣array-like,␣shape␣(n_features,␣n_hidden)
␣␣␣␣␣␣␣␣The␣input-to-hidden␣weights.

␣␣␣␣`intercept_hidden_`␣:␣array-like,␣shape␣(n_hidden,)
␣␣␣␣␣␣␣␣The␣bias␣added␣to␣the␣hidden␣layer␣neurons.
```

```
␣␣␣␣`coef_output_`␣:␣array-like,␣shape␣(n_hidden,␣n_outputs_)
␣␣␣␣␣␣␣␣The␣hidden-to-output␣weights.

␣␣␣␣`label_binarizer_`␣:␣LabelBinarizer
␣␣␣␣␣␣␣␣A␣LabelBinarizer␣object␣trained␣on␣the␣training␣set.

␣␣␣␣References
␣␣␣␣----------
␣␣␣␣Liang,␣Nan-Ying,␣et␣al.
␣␣␣␣␣␣␣␣"A␣fast␣and␣accurate␣online␣sequential␣learning␣algorithm␣for
␣␣␣␣␣␣␣␣feedforward␣networks."␣Neural␣Networks,␣IEEE␣Transactions␣on
␣␣␣␣␣␣␣␣17.6␣(2006):␣1411-1423.
␣␣␣␣␣␣␣␣http://www.ntu.edu.sg/home/egbhuang/pdf/OS-ELM-TNN.pdf

␣␣␣␣Zong,␣Weiwei,␣Guang-Bin␣Huang,␣and␣Yiqiang␣Chen.
␣␣␣␣␣␣␣␣"Weighted␣extreme␣learning␣machine␣for␣imbalance␣learning."
␣␣␣␣␣␣␣␣Neurocomputing␣101␣(2013):␣229-242.

␣␣␣␣Glorot,␣Xavier,␣and␣Yoshua␣Bengio.␣"Understanding␣the␣difficulty␣of
␣␣␣␣␣␣␣␣training␣deep␣feedforward␣neural␣networks."␣International␣Conference
␣␣␣␣␣␣␣␣on␣Artificial␣Intelligence␣and␣Statistics.␣2010.
␣␣␣␣"""
    def __init__(self, n_hidden=100, activation='relu', C=1,
                 class_weight=None, weight_scale=1.0, batch_size=None,
                 verbose=False, warm_start=False, random_state=None):
        super(ELMClassifier, self).__init__(n_hidden=n_hidden,
                                            activation=activation,
                                            C=C, class_weight=class_weight,
                                            weight_scale=weight_scale,
                                            batch_size=batch_size,
                                            verbose=verbose,
                                            warm_start=warm_start,
                                            random_state=random_state)

        self.label_binarizer_ = LabelBinarizer(-1, 1)

    def partial_fit(self, X, y, classes=None, sample_weight=None):
        """Fit␣the␣model␣to␣the␣data␣X␣and␣target␣y.

␣␣␣␣␣␣␣␣Parameters
␣␣␣␣␣␣␣␣----------
␣␣␣␣␣␣␣␣X␣:␣{array-like,␣sparse␣matrix},␣shape␣(n_samples,␣n_features)
␣␣␣␣␣␣␣␣␣␣␣␣The␣input␣data.

␣␣␣␣␣␣␣␣y␣:␣array-like,␣shape␣(n_samples,)
␣␣␣␣␣␣␣␣␣␣␣␣Subset␣of␣the␣target␣values.
```

```
        classes : array-like, shape (n_classes,)
            List of all the classes that can possibly appear in the y vector.

            Must be provided at the first call to partial_fit, can be omitted
            in subsequent calls.

        sample_weight : array-like, shape (n_samples,)
            Per-sample weights. Rescale C per sample. Higher weights
            force the classifier to put more emphasis on these points.

        Returns
        -------
        self : returns a trained elm usable for prediction.
        """
        self.classes_ = classes

        super(ELMClassifier, self).partial_fit(X, y, sample_weight)

        return self

    def decision_function(self, X):
        """Decision function of the elm model

        Parameters
        ----------
        X : {array-like, sparse matrix}, shape (n_samples, n_features)
            The input data.

        Returns
        -------
        y : array-like, shape (n_samples,) or (n_samples, n_classes)
            The predicted values.
        """
        y_scores = self._decision_scores(X)

        if self.n_outputs_ == 1:
            return y_scores.ravel()
        else:
            return y_scores

    def predict(self, X):
        """Predict using the ELM model

        Parameters
        ----------
        X : {array-like, sparse matrix}, shape (n_samples, n_features)
            The input data.
        Returns
        -------
        y : array-like, shape (n_samples,) or (n_samples, n_classes)
            The predicted classes, or the predicted values.
        """
        y_scores = self._decision_scores(X)

        return self.label_binarizer_.inverse_transform(y_scores)

    def predict_proba(self, X):
        """Probability estimates.

        Warning: the estimates aren't callibrated since the model optimizes a
        penalized least squares objective function based on the One Vs Rest
        binary encoding of the class membership.

        Parameters
        ----------
        X : {array-like, sparse matrix}, shape (n_samples, n_features)
            The input data.

        Returns
        -------
        y_prob : array-like, shape (n_samples, n_classes)
            The predicted probability of the sample for each class in the
            model, where classes are ordered as they are in
            `self.classes_`.
        """
        y_scores = self._decision_scores(X)

        if len(self.classes_) == 2:
            y_scores = logistic(y_scores)
            return np.hstack([1 - y_scores, y_scores])
        else:
            return softmax(y_scores)


class ELMRegressor(BaseELM, RegressorMixin):
    """Extreme learning machine regressor.

    The algorithm trains a single-hidden layer feedforward network by computing
    the hidden layer values using randomized parameters, then solving
    for the output weights using least-square solutions. For prediction,
    ELMRegressor computes the forward pass resulting in continuous output
    values.

    This implementation works with data represented as dense and sparse numpy
    arrays of floating point values for the features.
    Parameters
    ----------
    C : float, optional, default 100
        A regularization term that controls the linearity of the decision
        function. Smaller value of C makes the decision boundary more linear.

    weight_scale : float, default 1.
        Initializes and scales the input-to-hidden weights.
        The weight values will range between plus and minus
        'sqrt(weight_scale * 6. / (n_features + n_hidden))' based on the
        uniform distribution.

    n_hidden : int, default 100
        The number of units in the hidden layer.

    activation : {'logistic', 'tanh', 'relu'}, default 'relu'
        Activation function for the hidden layer.

         - 'logistic', the logistic sigmoid function,
            returns f(x) = 1 / (1 + exp(x)).

         - 'tanh', the hyperbolic tan function,
            returns f(x) = tanh(x).

         - 'relu', the rectified linear unit function,
            returns f(x) = max(0, x).

    batch_size : int, optional, default None
        If None is given, batch_size is set as the number of samples.
        Otherwise, it will be set as the given integer.

    verbose : bool, optional, default False
        Whether to print the training score.

    warm_start : bool, optional, default False
        When set to True, reuse the solution of the previous
        call to fit as initialization, otherwise, just erase the
        previous solution.

    random_state : int or RandomState, optional, default None
        State of or seed for random number generator.

    Attributes
    ----------
    `classes_` : array-list, shape (n_classes,)
        Class labels for each output.
```

```
␣␣␣␣‘n_outputs_‘␣:␣int
␣␣␣␣␣␣␣␣Number␣of␣output␣neurons.

␣␣␣␣‘coef_hidden_‘␣:␣array-like,␣shape␣(n_features,␣n_hidden)
␣␣␣␣␣␣␣␣The␣input-to-hidden␣weights.

␣␣␣␣‘intercept_hidden_‘␣:␣array-like,␣shape␣(n_hidden,)
␣␣␣␣␣␣␣␣The␣bias␣added␣to␣the␣hidden␣layer␣neurons.

␣␣␣␣‘coef_output_‘␣:␣array-like,␣shape␣(n_hidden,␣n_outputs_)
␣␣␣␣␣␣␣␣The␣hidden-to-output␣weights.

␣␣␣␣References
␣␣␣␣----------
␣␣␣␣Liang,␣Nan-Ying,␣et␣al.
␣␣␣␣␣␣␣␣"A␣fast␣and␣accurate␣online␣sequential␣learning␣algorithm␣for
␣␣␣␣␣␣␣␣feedforward␣networks."␣Neural␣Networks,␣IEEE␣Transactions␣on
␣␣␣␣␣␣␣␣17.6␣(2006):␣1411-1423.
␣␣␣␣␣␣␣␣http://www.ntu.edu.sg/home/egbhuang/pdf/OS-ELM-TNN.pdf

␣␣␣␣Zong,␣Weiwei,␣Guang-Bin␣Huang,␣and␣Yiqiang␣Chen.
␣␣␣␣␣␣␣␣"Weighted␣extreme␣learning␣machine␣for␣imbalance␣learning."
␣␣␣␣␣␣␣␣Neurocomputing␣101␣(2013):␣229-242.

␣␣␣␣Glorot,␣Xavier,␣and␣Yoshua␣Bengio.␣"Understanding␣the␣difficulty␣of
␣␣␣␣␣␣␣␣training␣deep␣feedforward␣neural␣networks."␣International␣Conference
␣␣␣␣␣␣␣␣on␣Artificial␣Intelligence␣and␣Statistics.␣2010.
␣␣␣␣"""
    def __init__(self, n_hidden=100, activation='relu', weight_scale=1.0,
                 batch_size=None, C=1, verbose=False, warm_start=False,
                 random_state=None):
        super(ELMRegressor, self).__init__(n_hidden=n_hidden,
                                           activation=activation,
                                           C=C, class_weight=None,
                                           weight_scale=weight_scale,
                                           batch_size=batch_size,
                                           verbose=verbose,
                                           warm_start=warm_start,
                                           random_state=random_state)
    def get_features(self, X, y):
        elm = super(ELMRegressor, self).fit(X, y)
        beta = elm.coef_output_

        # Apply the activation method
        activation = ACTIVATIONS[self.activation]
        features = safe_sparse_dot(X, beta.T);
        features = activation(features)
        return features

    def predict(self, X):
        y_pred = self._decision_scores(X)

        if self.n_outputs_ == 1:
            return y_pred.ravel()
        else:
            return y_pred
```

III. Utilities Files

```
from __future__ import division
import numpy as np
import pandas as pd
import gensim as gs
import nltk
import re
import random

from scipy import sparse
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics.pairwise import cosine_similarity
from sklearn import preprocessing
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.feature_selection import chi2
from sklearn.feature_selection import SelectKBest


def fair_accuracy(y_pred, y_test):
        """Compute weighted accuracy."""
        # get indices for each class
        sarcastic_indices = y_test == 1
        non_sarcastic_indices = y_test == -1

        # Fair Testing score
        score = accuracy_score(y_pred[sarcastic_indices],
                                                    y_test[sarcastic_indices]) + \
                accuracy_score(y_pred[non_sarcastic_indices],
                                        y_test[non_sarcastic_indices])
        return score / 2

def fair_recall( y_test, y_pred):
        """Compute recall."""

        count = len(y_test)
        tp_sar= 0
        tp_nonsar= 0
        fn_sar = 0
        fn_nonsar = 0
        for j in range(0,count):
            if y_test[j] == 1 and y_pred[j] == 1:
                tp_sar += 1
            if y_test[j] == 1 and y_pred[j] == -1:
                fn_sar += 1
            if y_test[j] == -1 and y_pred[j] == -1:
                tp_nonsar += 1
            if y_test[j] == -1 and y_pred[j] == 1:
                fn_nonsar += 1

        sarcastic = tp_sar / (tp_sar + fn_sar)
        nonSarcastic = tp_nonsar / (tp_nonsar + fn_nonsar)
        return (sarcastic + nonSarcastic)/ 2

def fair_precision( y_test , y_pred):
        """Compute precision."""
        count = len(y_test)
        tp_sar= 0
        tp_nonsar= 0
        fp_sar = 0
        fp_nonsar = 0
        for j in range(0,count):
            if y_test[j] == 1 and y_pred[j] == 1:
                tp_sar += 1
            if y_test[j] == -1 and y_pred[j] == 1:
                fp_sar += 1
            if y_test[j] == -1 and y_pred[j] == -1:
                tp_nonsar += 1
            if y_test[j] == 1 and y_pred[j] == -1:
                fp_nonsar += 1

        sarcastic = tp_sar / (tp_sar + fp_sar)
        nonSarcastic = tp_nonsar / (tp_nonsar + fp_nonsar)
        return (sarcastic + nonSarcastic)/ 2

def balance_dataset(X, y):
        """Balance dataset such that the number of sarcastic and non-sarcastic
            responses are equal.
        """
        # get indices for each class
        X_sarcastic = X[np.where(y == 1)]
        y_sarcastic = y[np.where(y == 1)]

        X_non_sarcastic = X[np.where(y == -1)]
        y_non_sarcastic = y[np.where(y == -1)]
        n_sarcastic = y_sarcastic.shape[0]


        # check if it's a sparse matrix
        if sparse.issparse(X):
                X = sparse.vstack([X_sarcastic, X_non_sarcastic[:n_sarcastic]])
        else:
                X = np.vstack([X_sarcastic, X_non_sarcastic[:n_sarcastic]])

        y = np.hstack([y_sarcastic, y_non_sarcastic[:n_sarcastic]])

        return X, y

def read_dataset(location):
        """Read␣dataset␣into␣X␣and␣y␣matrices."""
        # Read the 'qr_meta.csv' excel sheet containing the quote-response pairs
        qr = pd.read_csv(location + "qr_meta.csv", encoding='utf')

        # Read the 'qr_averages.csv' excel sheet containing the average sarcasm de
        sarcasm_table = pd.read_csv(location + "qr_averages.csv",
encoding='utf')

        # Join the two tables on the key column
        dataset = qr.merge(sarcasm_table, on='key')

        # Remove the rows where 'sarcasm' value is NaN
        dataset = dataset[pd.notnull(dataset['sarcasm'])]

        # Extract sarcasm labels
        y = np.array(dataset[['sarcasm']])

        # Combine the two columns quote and response into a single column
        dataset = dataset["quote"] + "␣" + dataset["response"]
        X = np.array(dataset)

        # Threshold y such that values above 0.5
        # are set to 1, and the rest to -1
        y[y >= 0.5] = 1
        y[y < 0.5] = -1

        return X, y

def analyze_dataset(X, y):
        """Report␣information␣about␣the␣dataset."""
        # get indices for each class
        sarcastic_indices = y == 1
        non_sarcastic_indices = y == -1
        print 'sarcastic', np.sum(sarcastic_indices)
        print 'non-sarcastic', np.sum(non_sarcastic_indices)

def analyze_output(y_pred, y_test):
        """Report information about the output."""
        # get indices for each class
        cm = confusion_matrix(y_test, y_pred)
        print 'Confusion matrix, '
        print cm

def FT_computeSim(dataset):
    #Setup LDA Model----------------------------------------------------
    documents = dataset["quote"] + " " + dataset["response"]

    #empirically set num topics
    numTopics = 40

    texts = [[word for word in document.lower().split()] \
    for document in documents]

    #Create Dictionary
    dictionary = gs.corpora.Dictionary(texts)
    dictionary.save('sarcasm.dict') # store the dictionary, for future reference

    # remove common words, words that appear only once and tokenize
    stoplist = nltk.corpus.stopwords.words('english')
    stop_ids = [dictionary.token2id[stopword] for stopword in stoplist \
    if stopword in dictionary.token2id]
    once_ids = [tokenid for tokenid, docfreq in dictionary.dfs.iteritems()\
    if docfreq == 1]
    dictionary.filter_tokens(stop_ids + once_ids)

    #create corpus
    corpus = [dictionary.doc2bow(text) for text in texts]
    # store to disk, for later use
    gs.corpora.MmCorpus.serialize('sarcasmCorpus.mm', corpus)

    lda = gs.models.LdaModel(corpus, id2word=dictionary, num_topics=numTopics, \
    update_every=1)

    # transform corpus to LDA space and index it
    index = gs.similarities.MatrixSimilarity(lda[corpus])
    index.save('sarcasm')

    topics = [lda[c] for c in corpus]

    quotes = [dictionary.doc2bow(quote.lower().split()) \
    for quote in dataset["quote"]]
    responses = [dictionary.doc2bow(response.lower().split()) \
    for response in dataset["response"]]

    #get topic distribution for all quotes and responses
    quoteTopics = lda[quotes]
    responseTopics = lda[responses]

    quoteTopicVectors = []
    responseTopicVectors = []

    for item in quoteTopics:
        quoteTopicVectors.append(item)

    for item in responseTopics:
        responseTopicVectors.append(item)

    qrSimilarity = [0]*len(quoteTopicVectors)

    #compute quote/response similarity
    for i in range(0,len(quoteTopicVectors)):
        iQuote = [0]*numTopics
        iResponse = [0]*numTopics
        for topic in quoteTopicVectors[i]:
            iQuote[topic[0]] = topic[1]
        for topic in responseTopicVectors[i]:
            iResponse[topic[0]] = topic[1]
        qrSimilarity[i] =cosine_similarity( iQuote, iResponse)[0][0]

    return qrSimilarity

def FT_firstWord(documents):
    #tokoenize document
    tokens = [nltk.word_tokenize(document.lower()) for document in documents]

    #create array of first words
    array = np.array([token[0] for token in tokens])

    #label and transform first words
    le = preprocessing.LabelEncoder()
    le.fit(array)
    array = le.transform(array)

    binArray = label2binary(array)

    return binArray

def FT_lastWord(documents):
    #tokoenize document
    tokens = [nltk.word_tokenize(document.lower()) for document in documents]

    #create array of first words
    array = np.array([token[len(token)-1] for token in tokens])

    #label and transform first words
    le = preprocessing.LabelEncoder()
    le.fit(array)
    array = le.transform(array)

    binArray = label2binary(array)

    return binArray

def FT_capitalsBaseline(documents):
    # number of capitals
    capitals = np.array([len(re.findall("[A-Z]", document)) \
    for document in documents])
    #normalize counts
    normalize(capitals)

    #number of words in all capitals
    allCapitals = np.array([len(re.findall("[A-Z][A-Z]+", document)) \
    for document in documents])
    #normalize counts
    normalize(allCapitals)

    caps = np.column_stack((capitals, allCapitals))

    return caps

def FT_wordOverlap(quote, response):
    #compute word overlap
    quoteTokens = [nltk.word_tokenize(q.lower()) for q in quote]
    responseTokens = [nltk.word_tokenize(r.lower()) for r in response]
    overlap = [numIntersect(responseTokens[i],quoteTokens[i])/\
    numUnion(responseTokens[i],quoteTokens[i]) for i in range(0, len(quoteTokens))]
    overlap = normalize(overlap)
    return overlap

def FT_punct(responses):
    quest = normalize([len(re.findall("\'", response)) for response in responses])
    peri = normalize([len(re.findall(".", response))for response in responses])
    comma = normalize([ len(re.findall(",", response)) for response in responses])

    stack = np.column_stack((quest, peri))
    stack = np.column_stack((stack, comma))
    return stack

def FT_respLength(responses):
    length = normalize([len(response.split()) for response in responses])

    return length

def FT_punctBaseline(responses):
    #pulls baseline punctuation features
    # based on Davidov et. al 2010
    exclam = normalize([len(re.findall("!", response)) for response in responses])
    quest = normalize([len(re.findall("\?", response)) for response in responses])
    quote = normalize([len(re.findall("\"", response))for response in responses])

    stack = np.column_stack((exclam, quest))
    stack = np.column_stack((stack, quote))

    return stack

def FT_TFIDF(document, sarcasm):
    sarcasm = sarcasm.flatten()
    transformer = TfidfVectorizer()
    TFIDF = transformer.fit_transform(document)
    TFIDF = SelectKBest(chi2, k=1500).fit_transform(TFIDF, sarcasm)

    TFIDF = TFIDF.toarray()

    return TFIDF

def normalize(counts):
    #normalize counts in input vector
    maxCount = max(counts)
    if maxCount>0 :
        norm = [count/maxCount for count in counts]
    else:
        norm = counts
    return norm

def label2binary(array):
    #Convert to binary
    maxLabel = max(array)
    maxLength = len("{0:b}".format(maxLabel))
    binFormat = '0' + str(maxLength) + 'b'
    df = [format(num, binFormat) for num in array]
    stack = np.array([int(d[0]) for d in df])
    for i in range(1,maxLength):
        stack = np.column_stack((stack,[int(d[i]) for d in df]))
    return stack
```

```
def numIntersect(a, b):
     return len(list(set(a) & set(b)))

def numUnion(a, b):
     return len(list(set(a) | set(b)))

def read_dataset_whole(location):
        # Read the 'qr_meta.csv' excel sheet containing the quote-response pairs
        qr = pd.read_csv(location + "qr_meta.csv", encoding='utf')

        # Read the 'qr_averages.csv' excel sheet containing the average sarcasm deg
        sarcasm_table = pd.read_csv(location + "qr_averages.csv",
encoding='utf')

        # Join the two tables on the key column
        dataset = qr.merge(sarcasm_table, on='key')

        # Remove the rows where 'sarcasm' value is NaN
        dataset = dataset[pd.notnull(dataset['sarcasm'])]

        # Extract sarcasm labels
        y = np.array(dataset[['sarcasm']])

        # Threshold y such that values above 0.5
        # are set to 1, and the rest to -1
        y[y >= 0.5] = 1
        y[y < 0.5] = -1

        return dataset, y

def rdm_data_split(y):
    # Get indices of sarcastic and non-sarcastic
    sarcastic_indices = [i for i,x in enumerate(y) if x == 1]
    non_sarcastic_indices = [i for i,x in enumerate(y) if x == -1]

    #randomly assign half sarcastic comments to test/train
    random.shuffle(sarcastic_indices)
    random.shuffle(non_sarcastic_indices)

    train_idx = np.hstack((sarcastic_indices[:len(sarcastic_indices)//2],
    non_sarcastic_indices[:len(sarcastic_indices)//2] ))
    test_idx = np.hstack((sarcastic_indices[len(sarcastic_indices)//2:],
    non_sarcastic_indices[len(sarcastic_indices)//2:] ))

    return train_idx, test_idx

def getFeatures(dataset, y):
    FT = []
    FT.append(FT_computeSim(dataset))
    FT.append(FT_firstWord(dataset['response']))
    FT.append(FT_lastWord(dataset['response']))
    FT.append(FT_respLength(dataset['response']))
    FT.append(FT_punct(dataset['response']))
    FT.append(FT_punctBaseline(dataset['response']))
    FT.append(FT_capitalsBaseline(dataset['response']))
    FT.append(FT_TFIDF(np.array(dataset["quote"] + "␣" + dataset["response"]), y))

    #vstack features into matrix
    stack = np.column_stack((FT[0], FT[1]))
    for i in range(2, len(FT)):
        stack = np.column_stack((stack, FT[i]))

    stack = SelectKBest(chi2, k=1500).fit_transform(stack, y)

    return stack

def getBaselineFeatures(dataset):
    FT = []
    FT.append(FT_respLength(dataset['response']))
    FT.append(FT_capitalsBaseline(dataset['response']))
    FT.append(FT_punctBaseline(dataset['response']))

    #vstack features into matrix
    stack = np.column_stack((FT[0], FT[1]))
    return stack

def splitFeatures(allFeatures, y, train_idx, test_idx):

    Xtrain = np.array([allFeatures[i] for i in train_idx])
    Xtest = np.array([allFeatures[i] for i in test_idx])
    ytrain = np.array([y[i] for i in train_idx]).flatten()
    ytest = np.array([y[i] for i in test_idx]).flatten()

    return Xtrain, ytrain, Xtest, ytest

from scipy import sparse
import numpy as np
import pandas as pd
import nltk
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
import gensim
from gensim.models.ldamodel import LdaModel
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from scipy.sparse import *
from extreme_learning_machines import ELMRegressor, ELMClassifier
from scipy.sparse import issparse
def balance_dataset(X, y):
        """Balance␣dataset␣such␣that␣the␣number␣of␣sarcastic␣and␣non-sarcastic
␣␣␣␣␣␣␣␣␣␣␣responses␣are␣equal.
␣␣␣␣␣␣␣␣"""
        # get indices for each class
        X_sarcastic = X[np.where(y == 1)]
        y_sarcastic = y[np.where(y == 1)]

        X_non_sarcastic = X[np.where(y == -1)]
        y_non_sarcastic = y[np.where(y == -1)]

        n_sarcastic = y_sarcastic.shape[0]


        # check if it's a sparse matrix
        if sparse.issparse(X):
                X = sparse.vstack([X_sarcastic, X_non_sarcastic[:n_sarcastic]])
        else:
                X = np.vstack([X_sarcastic, X_non_sarcastic[:n_sarcastic]])

        y = np.hstack([y_sarcastic, y_non_sarcastic[:n_sarcastic]])

        return X, y
def fair_accuracy(y_pred, y_test):
        """Compute␣weighted␣accuracy."""
        # get indices for each class
        sarcastic_indices = y_test == 1
        non_sarcastic_indices = y_test == -1

        # Fair Testing score
        score = accuracy_score(y_pred[sarcastic_indices],
                                                          y_test[sarcastic_indices]) + \
                accuracy_score(y_pred[non_sarcastic_indices],
                                            y_test[non_sarcastic_indices])
        return score / 2

def dumbo(y_pred, y_test):
        """To␣test␣scoring."""
        return 1

def extract_ELM_features(X, y, with_feature_extraction=False, \
                                                                  chi_square = 3000, n_hid

        #transformer = TfidfVectorizer(stop_words="english", sublinear_tf =True)
        #X = transformer.fit_transform(X)
        n_features = X.shape[1]
        k = np.min([3000, n_features])

        X = SelectKBest(chi2, k=k).fit_transform(X, y)
        X, y = balance_dataset(X, y)

        if with_feature_extraction:
                reg = ELMRegressor(n_hidden=n_hidden, weight_scale=25, activation='
                                                random_state=0)
                #reg = ELMRegressor(n_hidden=50, weight_scale=5, activation='relu',
                if issparse(X):
                        X = reg.get_features(X, X.todense())
                else:
                        X = reg.get_features(X, X)

                #X = np.hstack([X_, X])
                #chi_square = np.min([chi_square, n_features])
        chi_square = np.min([chi_square, X.shape[1]])
        X = SelectKBest(chi2, k=chi_square).fit_transform(X, y)
        #print X.shape
        #X, y = balance_dataset(X, y)
        return X, y

def extract_ELM_features_response_quote(X_quotes, X_responses, y):
        transformer = TfidfVectorizer()
        X_quotes = transformer.fit_transform(X_quotes)
        X_responses = transformer.fit_transform(X_responses)

        X_quotes, y = balance_dataset(X_quotes, y)
        X_responses, y = balance_dataset(X_responses, y)

        X_quotes = SelectKBest(chi2, k=5000).fit_transform(X_quotes, y)
        X_responses = SelectKBest(chi2, k=2000).fit_transform(X_responses, y)

        reg = ELMRegressor(n_hidden=1000, weight_scale=1, activation='relu', rando
        X = reg.get_features(X_responses, X_responses.todense())
        X = X_responses

        return X, y

def lda(X, y):
        """Extract␣features␣using␣lda.

␣␣␣␣␣␣␣␣Input:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣X:␣Corpus
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣y:␣target␣labels
```

```
␣␣␣␣␣␣␣␣"""
        X = CountVectorizer(stop_words='english').fit_transform(X)
        X = SelectKBest(chi2, k=1500).fit_transform(X, y)
        X = coo_matrix(X)

        corpus = gensim.matutils.Sparse2Corpus(X.T)

        lda = LdaModel(corpus, num_topics=50,
                   update_every=0, passes=50, decay=0.8, chunksize=9000)
        return gensim.matutils.corpus2csc(lda[corpus]).T

def tokenize_matrix(X):
        """Extract␣a␣tokenized␣version␣of␣the␣matrix."""
        max_length = 0
        n_samples = X.shape[0]
        for i in range(n_samples):
                max_length = max(max_length, len(nltk.word_tokenize(X[i])))

        X_ = np.zeros((n_samples, max_length), dtype='object')

        for i in range(n_samples):
                sentence = nltk.word_tokenize(X[i])
                token_length = len(sentence)
                X_[i, :token_length] = sentence

        return X_

def extract_tfidf_on_POS(X):
        """Extracting␣tfidf␣features␣on␣POS."""
        def tokenize(text):
            tmp = nltk.pos_tag(nltk.word_tokenize(text))
            return [tag for j, tag in tmp]

        #this can take some time
        tfidf = TfidfVectorizer(tokenizer=tokenize, stop_words='english')
        X = tfidf.fit_transform(X)

        return X

def read_dataset(location):
        """Read␣dataset␣into␣X␣and␣y␣matrices."""
        # Read the 'qr_meta.csv' excel sheet containing the quote-response pairs
        qr = pd.read_csv(location + "qr_meta.csv", encoding='utf')

        # Read the 'qr_averages.csv' excel sheet containing the average sarcasm de
        sarcasm_table = pd.read_csv(location + "qr_averages.csv",
encoding='utf')
```

```
        # Join the two tables on the key column
        dataset = qr.merge(sarcasm_table, on='key')

        # Remove the rows where 'sarcasm' value is NaN
        dataset = dataset[pd.notnull(dataset['sarcasm'])]

        # Extract sarcasm labels
        y = np.array(dataset[['sarcasm']])

        # Combine the two columns quote and response into a single column
        dataset = dataset["quote"] + "␣" + dataset["response"]
        X = np.array(dataset)

        # Threshold y such that values above 0.5
        # are set to 1, and the rest to -1 \
        y[y >= 0.5] = 1
        y[y < 0.5] = -1

        return X, y

def read_dataset_csv(location):
        """Read␣dataset␣into␣X␣and␣y␣matrices."""
        # Read the 'qr_meta.csv' excel sheet containing the quote-response pairs
        qr = pd.read_csv(location + "qr_meta.csv", encoding='utf')

        # Read the 'qr_averages.csv' excel sheet containing the average sarcasm de
        sarcasm_table = pd.read_csv(location + "qr_averages.csv",
encoding='utf')

        # Join the two tables on the key column
        dataset = qr.merge(sarcasm_table, on='key')

        # Remove the rows where 'sarcasm' value is NaN
        dataset = dataset[pd.notnull(dataset['sarcasm'])]

        # Extract sarcasm labels
        y = np.array(dataset[['sarcasm']])

        # Combine the two columns quote and response into a single column
        dataset = dataset["quote"] + "␣" + dataset["response"]
        #X = np.array(dataset)

        # Threshold y such that values above 0.5
        # are set to 1, and the rest to -1 \
        y[y >= 0.5] = 1
        y[y < 0.5] = -1

        return dataset, y
```

```
def read_quote_response_independently(location):
        """Read␣dataset␣into␣X␣and␣y␣matrices."""
        # Read the 'qr_meta.csv' excel sheet containing the quote-response pairs
        qr = pd.read_csv(location + "qr_meta.csv", encoding='utf')

        # Read the 'qr_averages.csv' excel sheet containing the average sarcasm deg
        sarcasm_table = pd.read_csv(location + "qr_averages.csv",
encoding='utf')

        # Join the two tables on the key column
        dataset = qr.merge(sarcasm_table, on='key')

        # Remove the rows where 'sarcasm' value is NaN
        dataset = dataset[pd.notnull(dataset['sarcasm'])]

        # Extract sarcasm labels
        y = np.array(dataset[['sarcasm']])

        # Combine the two columns quote and response into a single column
        X_quotes =  np.array(dataset["quote"])
        X_responses =  np.array(dataset["response"])

        # Threshold y such that values above 0.5
        # are set to 1, and the rest to -1 \
        y[y >= 0.5] = 1
        y[y < 0.5] = -1

        return X_quotes, X_responses, y

def analyze_dataset(X, y):
        """Report␣information␣about␣the␣dataset."""
        # get indices for each class
        sarcastic_indices = y == 1
        non_sarcastic_indices = y == -1

        print 'sarcastic', np.sum(sarcastic_indices)
        print 'non-sarcastic', np.sum(non_sarcastic_indices)

def analyze_output(y_pred, y_test):
        """Report␣information␣about␣the␣output."""
        # get indices for each class
        cm = confusion_matrix(y_test, y_pred)
        print 'Confusion␣matrix,␣'
        print cm
```