IDENTIFYING DUPLICATE QUESTIONS

Capstone Project 2: Milestone Report Siri Surabathula

Problem Statement

Question-answering and knowledge-sharing platforms like Quora and Stack-Exchange require mechanisms to group questions in their database based on similar intent. According to Quora about 100 million users visit their website every month, and a lot of the questions end up being duplicates (duplicate questions could be differently worded but they have the same intent)
For example, the questions ‘What is the best way to travel from Houston to Atlanta ?’ and ‘Should I drive, fly or take a bus from Houston to Atlanta ?’, are duplicates.
User experience is greatly improved if Quora is able to identify duplicate questions as this enables them to find questions that have already been answered and also avoids the need to answer the same question multiple times.

Description of Data and Data-Processing

Source : Kaggle (Quora Question Pairs)

Description : The dataset contains a human-labeled training set and a test set. Each record in the training set represents a pair of questions with the text of both questions and a binary label which indicates whether the question-pair represents a duplicate or not.

Pre-processing

This dataset was processed by the ipython file tfidf_word2vec

Following preprocessing was performed on the text data using the gensim API

(gensim.parsing.preprocessing.preprocess_string) :

1. Tag Removal - Removed tags. For eg. "<i>Hello</i> <b>World</b>!" was converted to "Hello World!"
2. Repeating Whitespace Removal - Removed repeating whitespace characters (spaces, tabs, line breaks) and turned tabs & line breaks into spaces
3. Stopwords Removal - Removed stopwords. (the default stopwords list from gensim was used)
4. Case conversion and Stemming - Transformed text to lowercase and performed porter stemming.

Word2Vec Embedding

The text data was embedded in 300 dimensional space using two methods:

1. Word2Vec - Fasttext pretrained model for English (trained on Common Crawl and Wikipedia) was transfer-trained with the questions in training data using the gensim method intersect_word2vec_format. This method allows one to initialize a word2vec model with a vocabulary same as that of the training data, then intersects this vocabulary with the pretrained model. No words are added to the existing vocabulary, but intersecting words adopt the weights of the pretrained model, while non-intersecting words are left alone.
2. GloVe - Spacy pretrained model for English (trained on Common Crawl using GloVe). Provision was made for Out Of Vocabulary (OOV) words by randomly mapping each OOV word to one of 50 randomly generated vectors.

TF-IDF Weights

The term frequency-inverse document frequency (TF-IDF) weight was computed for each token in each question in the entire dataset using the scikit-learn API (TFIDFVectorizer). The Vectorizer was trained using the training set only. These weights were used later during the feature extraction process to compute weighted averages of the various pairwise distance and similarity metrics between the two question matrices for each row of data.

Feature Extraction

Pairwise Distance Metrics

The following pairwise distance and similarity metrics were computed. Each metric was computed for every combination of token-pairs formed by picking the first token (t_1) from question 1 and the second token (t_2) from question 2:

1. Chebyshev Distance - which is defined as,

   $$max|t_1 - t_2|$$

2. Bray-Curtis Distance - which is defined as,

   $$\sum |t_1 - t_2| \;/\; \sum |t_1 + t_2|$$

3. Cosine Distance - which is defined as,

   $$1 - \frac{t_1.t_2}{||t_1||_2 \; ||t_2||_2}$$

4. Correlation Distance - which is defined as,

$$1 - \frac{(t_1 - \bar{t}_1).(t_2 - \bar{t}_2)}{\|t_1 - t_1\|_2 \ \|t_2 - t_2\|_2}$$

5. Canberra Distance - which is defined as,

$$\sum \frac{|t_1 - t_2|}{|t_1| - |t_2|}$$

6. Cityblock Distance - which is defined as,

$$\sum |t_1 - t_2|$$

7. Euclidean Distance - which is defined as,

$$\|t_1 - t_2\|_2$$

8. L1 Distance - which is defined as,

$$\|t_1 - t_2\|_1$$

9. Minkowski Distance - which is defined as,

$$\sqrt[p]{\sum |t_1 - t_2|^p}$$

10. Squared Euclidean Distance - which is defined as,

$$\|t_1 - t_2\|_2^2$$

Weighted Norm of Pairwise Distance Matrices

For all the pairwise metrics computed above, the weighted average was computed using the TF-IDF weights extracted earlier for each token.

Document-2-Vector : Weighted Centroid for each Question

For each question, the weighted mean of all the token-vectors was computed using the TF-IDF weights

Parallel Processing for Feature Computation

Dask Delayed was used to compute the features on the train and test datasets as all these computations involved embarrassingly parallel operations.

Dask Delayed is useful when there clearly exists scope for parallelism in a problem (for example one or more functions can be evaluated independently), but it's not possible to convert the data structures involved into a big array or big DataFrame for computation.

In this case, the data structures involved (the set of arrays corresponding to question 1 as well as the set of arrays corresponding to question 2) were each numpy arrays where each row again contained a numpy array of varying size. These could not be converted to numpy arrays with fixed number of dimensions or to DataFrames with fixed number of columns. (this was ultimately because of the fact that the number of tokens varied from question to question)

Dask Delayed allows functions to be operated lazily. Rather than executing a function immediately, it defers execution, placing the function and its arguments into a task graph. Once the task graph of the entire computation is constructed, Dask schedulers execute the tasks in way that exploits all possibility of parallelism, which in turn leads to improvement in performance.

Fuzzy-Wuzzy Distance Metrics

The python library fuzzy-wuzzy was used to compute the following metrics:

1. Simple Ratio - which computes the similarity between two strings (in this case the two questions) using the simple edit distance (using Levenshtein distance) between the two strings

2. Partial Ratio - which improves on the simple ratio method above using a heuristic called "best partial" which is useful when the two strings are of noticeably different lengths. If the shorter string is length m, and the longer string is length n, the score of the best matching length-m substring is taken into account.

3. Token Sort Ratio - which involves tokenizing each string in question, sorting the tokens alphabetically, and then joining them back. These new strings are then compared using the simple ratio method.

4. Token Set Ratio - which involves tokenizing both the strings in question, but instead of immediately sorting and comparing, the tokens in the strings are split into three groups: the intersection component common to both strings and the two remainder components from each string. These sets are then used to perform the comparison. The scores increase when the intersection component makes up a larger percentage of the full string, and when the string remainders are more similar.

Modeling

The modeling was carried out for the entire dataset from (Quora Question Pairs)

Train-Test Split

The data was split into training and hold-out sets using sklearn.model_selection.train_test_split with a test size of 33%.

Logistic Regression

The processing steps are at tfidf_word2vec (section Modeling)

The scikit-learn API (sklearn.linear_model.LogisticRegression) was used.

Logistic regression is a linear model for classification. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

As an optimization problem, binary class L2 penalized logistic regression minimizes the following cost function,

$$min_{\omega,c} \frac{1}{2}\omega^T \omega + C \sum_{i=1}^{n} log\left(exp\left(-y_i\left(X_i^T \omega + c\right)\right) + 1\right)$$

The hyperparameters of the model are:

1. C (Inverse of regularization strength)
2. tolerance (Tolerance for stopping criteria)

Model Selection

5-fold cross-validation was used with scikit-learn API (sklearn.model_selection.RandomizedSearchCV) to find the optimum combination of hyperparameters.

The optimum parameters are as follows,

C (Inverse of regularization strength)	100
tolerance (Tolerance for stopping criteria)	1e-05

Prediction

The following is the classification report,

	precision	**recall**	**f1-score**	**support**
not duplicate	0.79	0.83	0.81	84248

duplicate	0.68	0.61	0.65	49146
micro avg	0.75	0.75	0.75	133394
macro avg	0.74	0.72	0.73	133394
weighted avg	0.75	0.75	0.75	133394

Model Accuracy Score

The accuracy, recall and prediction scores for the model were computed as follows:

accuracy score	0.7526
precision score	0.6836
recall score	0.6118

XGBoost

The processing steps are at tfidf_word2vec (section Modeling)

The XGBoost API was used.

XGBoost (Extreme Gradient Boosting), is an implementation of the Gradient Boosting technique introduced in the paper Greedy Function Approximation: A Gradient Boosting Machine, by Friedman.

XGBoost can be considered a special case of gradient descent in functional space,

$\widehat{y}_i = \sum_{k=1}^{K} f_k(x_i)$ where $f_k \in F$ F being the space of functions of all trees

The objective function to be minimized is given by,

$Obj = \sum_{i=1}^{n} l(y_i, \widehat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$ where l is the training loss and Ω the tree complexity

The prediction at time-step t is given by,

$$\widehat{y}_i^{(t)} = \widehat{y}_i^{(t-1)} + f_t(x_i)$$

The objective at time-step t is given by,

$$Obj^{(t)} = \sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(t-1)} + f_t(x_i)) + \sum_{i=1}^{t} \Omega(f_i) + constant \simeq \sum_{i=1}^{n} \left[l(y_i, \widehat{y}_i^{(t-1)}) + g_i f_t(x_i) + \tfrac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) + constant$$

where $g_i = \partial_{\widehat{y}_i^{(t-1)}} l(y_i, \widehat{y}_i^{(t-1)})$, $h_i = \partial^2_{\widehat{y}_i^{(t-1)}} l(y_i, \widehat{y}_i^{(t-1)})$

If the tree function is defined as follows,

$f_t(x) = w_{q(x)}$ where w is the weight of the j th leaf and $q(x)$ the mapping from x to leaf j

And the tree complexity is defined as follows,

$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$ where T is the total number of leaves and w_j is the weight of the j th leaf

Then the minimum value of the objective at t becomes,

$Obj^{(t)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j+\lambda} + \gamma T$ where the optimal leaf weight $w_j^* = -\frac{G_j}{H_j+\lambda}$

The algorithm now finds the best possible tree using the above optimum value of the objective which can also be considered the score of the tree (analogous to the gini-value score in decision trees). However, there could be infinite number of possible trees at any time-step t. To get over this, the algorithm adopts a greedy approach and calculates the gain in objective at the split level (instead of the entire tree) and thus finds the next optimum split that minimizes the objective. It continues to make splits until the complexity or regularization factor for the total number of leaves in the tree, γ, makes it infeasible for making any more splits.

The hyperparameters of the model are:

1. n_estimators(Number of boosted trees to fit)
2. reg_lambda(L2 regularization term on weights)
3. max_depth(Maximum tree depth for base learners)
4. learning_rate(Boosting learning rate (xgb's "eta"))
5. gamma(Min. loss to make a partition on a leaf)

Model Selection

5-fold cross-validation was used with scikit-learn API (sklearn.model_selection.RandomizedSearchCV) to find the optimum combination of hyperparameters.

The optimum parameters are as follows,

N_estimators (Number of boosted trees to fit)	200
Reg_lambda (L2 regularization term on weights)	6.31947
Max_depth (Maximum tree depth for base learners)	29
Learning_rate (Boosting learning rate (xgb's "eta"))	0.23
Gamma (Min. loss to make a partition on a leaf)	0.23

Prediction

The following is the classification report,

	precision	**recall**	**f1-score**	**support**
not duplicate	0.87	0.88	0.87	84248
duplicate	0.79	0.76	0.78	49146

micro avg	0.84	0.84	0.84	133394
macro avg	0.83	0.82	0.83	133394
weighted avg	0.84	0.84	0.84	133394

Model Accuracy Score

The accuracy, recall and prediction scores for the model were computed as follows:

accuracy score	0.8383
precision score	0.7647
recall score	0.7898

Deep Learning with Decomposable Attention Model

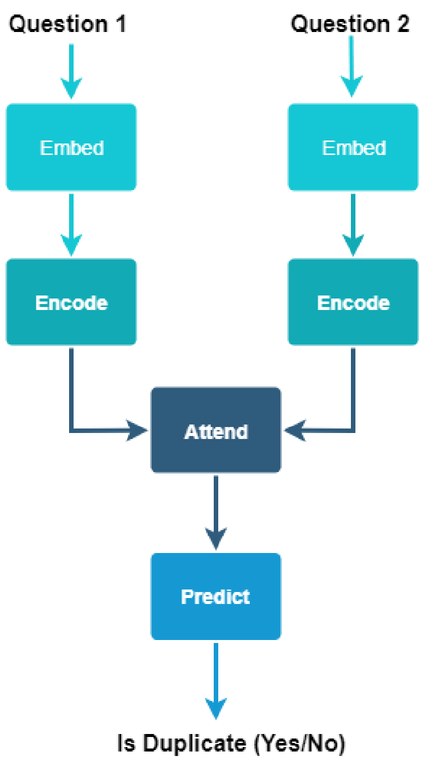
The processing steps are at DL_encode_attend (section Method 1)

This method was adapted from the section Example 1: A Decomposable Attention Model for Natural Language Inference from a blog by Matthew Honnibal.

The blog example itself is adapted from A Decomposable Attention Model for Natural Language Inference by Parikh et al. where the problem of predicting a class label over a pair of sentences, where the class represents the logical relationship between them, is tackled using attention mechanism. A crucial advantage is provided by the fact that sentence-to-vector reduction operates over the pair of sentences jointly, as opposed to encoding the sentences into vectors independently.

The Keras API with TensorFlow backend was used. A single GPU (Nvidia GeForce GTX 980 Ti) with 5.2 compute capability and 6GB memory was used for computation.

Keras provides a high-level Neural Network Python API that runs on top of high performance numerical computation libraries like TensorFlow and Theano. It provides a modular and extensible api with standalone and customizable modules for neural layers, cost functions, optimizers, initialization schemes, activation functions and regularization schemes which can be combined in different configurations to create new models.



The layers of the model are:

1. Embed - Convert tokens from text to vector space. A map for token ID to word vector (based on spacy's pretrained 300-dimensional word vectors trained on Common Crawl with GloVe) is created before-hand, and each question is converted to a sequence of token IDs outside the model. The model then takes the questions as sequences of token IDs and the ID-vector map as input.
2. Encode - Normally, an encode layer would have aggregated the sequence of vectors in each question into a single question matrix. However, this encode layer performs soft alignment (Parikh et al. sec 3.1) with tokens in the other question to compute attention weights for the subsequent attention layer. The weights are computed using the dot product (pairwise cosine distance) of the vectors of question a with the vectors of question b as follows (here $\overline{a_i}$ is the ith token vector of question a and $\overline{b_j}$ is the jth token vector of question b, e_{ij} is the unnormalized attention weight between $\overline{a_i}$ and $\overline{b_j}$, F is the attention-based encoding function and F' is the dot product of this function):

   $$e_{ij} = F'(\overline{a_i}, \overline{b_j}) = F(\overline{a_i})^T F(\overline{b_j})$$

   Here F is a feed-forward neural network with ReLU activations (Parikh et al. sec 3.1)

   Next, these weights are normalized as follows (here β_i is the sub-phrase of question b that is soft-aligned with the ith token $\overline{a_i}$ of question a and α_j is the sub-phrase of question a that is soft-aligned with the jth token b_j of question b, l_b and l_a are the total tokens in question b and question a respectively)

   $$\beta_i = \sum_{j=1}^{l_b} \frac{exp(e_{ij})}{\sum_{k=1}^{l_b} exp(e_{ik})} \overline{b_j}$$

$$\alpha_j = \sum_{i=1}^{l_a} \frac{exp(e_{ij})}{\sum_{k=1}^{l_a} exp(e_{kj})} \overline{a_i}$$

3. Attend - This step combines the above encoded matrices and reduces them to a single vector. This first involves a comparison step as follows,

   $$v_{1i} = G([\overline{a_i}, \beta_i])$$

   $$v_{2j} = G([\overline{b_j}, \alpha_j])$$

   Here G is again a feed-forward neural network with ReLU activations and $[.,.]$ represents the concatenation of two vectors.

   Next, the two sets of comparison vectors are aggregated by summation,

   $$v_1 = \sum_{i=1}^{l_a} v_{1i}$$

   $$v_2 = \sum_{j=1}^{l_b} v_{2j}$$

4. Predict / Classify - This involves the final feed-forward network classifier H as follows

   $$\hat{y} = H([v_1, v_2])$$

   the predicted class is given by $\hat{y} = argmax_i(\hat{y}_i)$

The cross-entropy loss for the model is given by,

$$L(\theta_F, \theta_G, \theta_H) = \frac{1}{N} \sum_{n=1}^{N} \sum_{c=1}^{C} y_c^{(n)} log \frac{exp(\hat{y}_c)}{\sum_{c'=1}^{C} exp(\hat{y}_{c'})}$$

The learnable parameters of the model are:

$\theta_F, \theta_G, \theta_H$ the learnable parameters of the feed-forward networks F, G and H respectively

The hyperparameters of the model are:

1. Dropout regularization rate for network F
2. Dropout regularization rate for network G
3. Dropout regularization rate for network H
4. Optimization method (Adam / RMSProp / Nadam / SGD)
5. Learning rate for the optimization method
6. Number of samples (batch size) per gradient update
7. The numbers of epochs used for training

Prediction

The following is the classification report,

	precision	**recall**	**f1-score**	**support**

not duplicate	0.86	0.84	0.85	84267
duplicate	0.74	0.76	0.75	49148
micro avg	0.81	0.81	0.81	133415
macro avg	0.80	0.80	0.80	133415
weighted avg	0.81	0.81	0.81	133415

Model Accuracy Score

The accuracy, recall and prediction scores for the model were computed as follows:

accuracy score	0.8125
precision score	0.7616
recall score	0.7379

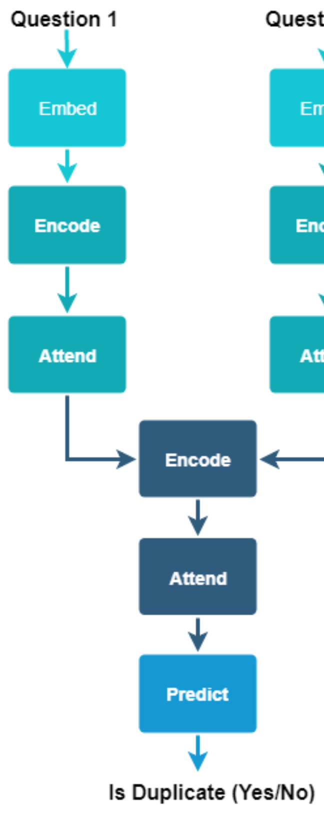
Deep Learning with Hierarchical Attention Model

The processing steps are at DL_encode_attend (section Method 2)

This method was adapted from the section Example 2: Hierarchical Attention Networks for Document Classification from the same blog as the previous method, by Matthew Honnibal.

The blog example itself is adapted from Hierarchical Attention Networks for Document Classification by Yang et al. where the problem of classification of documents is tackled using attention mechanism. The method uses attention to reduce a matrix to a vector. It does this by learning context vectors for the two attention steps (one for words and one for sentences), which is analogous to learning words or sentences from a document that summarize it. Applying the word and sentence context-vector analogy to a generic NLP problem, the attention steps can be seen as an effective feature extraction process, which in the case of duplicate question identification, can be employed to learn similarity between words and sentences belonging to a pair of questions.

Again Keras API with TensorFlow backend was used.



The layers of the model are:

1. Embed - Convert tokens from text to vector space. A map for token ID to word vector (based on spacy's pretrained 300-dimensional word vectors trained on Common Crawl with GloVe) is created before-hand, and each question is converted to a sequence of token IDs outside the model. The model then takes the questions as sequences of token IDs and the ID-vector map as input.
2. Encode Words - This uses a bidirectional GRU to encode each word into a context-specific vector. For every token vector x_{it} (embedded word vector for the word w_{it}) a forward GRU encodes the forward hidden state $\overrightarrow{h_{it}}$ and a backward GRU encodes the backward hidden state $\overleftarrow{h_{it}}$. A bidirectional wrapper layer then concatenates the forward and backward hidden states to summarize the information of the whole sentence centered around the word as h_{it}:

   $x_{it} = W_e w_{it}$ (here W_e is the embedding matrix)

   $\overrightarrow{h_{it}} = \overrightarrow{GRU}(x_{it})$

   $\overleftarrow{h_{it}} = \overleftarrow{GRU}(x_{it})$

   $h_{it} = [\overrightarrow{h_{it}}, \overleftarrow{h_{it}}]$

   Here GRU is a recurrent neural network that uses a gating mechanism to track the state of sequences without using separate memory cells (sec 2.1 of Yang et al.)
3. Word Attention - This step extracts the important words that are representative of the meaning of the sentence (or question) by using attention mechanism and computes a single vector for

each question by giving higher weights to the more important words. First h_{it} (the encoded vector of every word w_{it}) is converted to a hidden representation u_{it} by passing it through a Multilayer Perceptron (MLP) with one hidden layer and the tanh activation function with learned parameters W_w and b_w,

$$u_{it} = tanh(W_w h_{it} + b_w)$$

Next, the normalized importance of the word w_{it} is measured as the normalized similarity of its hidden vector u_{it} with the context vector u_w using the softmax function (the context vector u_w is a randomly initialized learned parameter of the attention layer),

$$\alpha_{it} = \sum_t \frac{exp(u_{it}^T u_w)}{\sum_t exp(u_{it}^T u_w)}$$

Now, a single sentence vector is computed for each question by aggregating the vectors h_{it} and the weights α_{it} with summation as follows,

$$s_i = \sum_t \alpha_{it} h_{it}$$

5. Encode Sentences - This uses a bidirectional GRU to encode each sentence (question) into a context-specific vector. For every sentence vector s_i (single sentence vector computed in the previous step for every question) a forward GRU encodes the forward state $h_i^{\rightarrow}$ and a backward GRU encodes the backward hidden state $h_i^{\leftarrow}$. A bidirectional wrapper layer then concatenates the forward and backward hidden states to summarize the information of the set of sentences centered around the sentence as h_i. Unlike the case of document classification where bidirectional GRU may deal with any number of sentence vectors at a time, the duplicate question problem deals with only two sentence (question) vectors at a time:

$$h_i^{\rightarrow} = GRU^{\rightarrow}(s_i)$$

$$h_i^{\leftarrow} = GRU^{\leftarrow}(s_i)$$

$$h_i = [h_i^{\rightarrow}, h_i^{\leftarrow}]$$

6. Sentence Attention - This step extracts the important features that are representative of the relationship between the two sentences (or questions) by using attention mechanism. It computes a single vector for both questions by giving higher weights to the more important features. First h_i (the encoded vector of every sentence s_i) is converted to a hidden representation u_i by passing it through a Multilayer Perceptron (MLP) with one hidden layer and the tanh activation function with learned parameters W_s and b_s,

$$u_i = tanh(W_s h_i + b_s)$$

Next, the normalized importance of the sentence s_i is measured as the normalized similarity of its hidden vector u_i with the context vector u_s using the softmax function (the context vector u_s is a randomly initialized learned parameter of the attention layer),

$$\alpha_i = \sum_i \frac{exp(u_i^T u_s)}{\sum_i exp(u_i^T u_s)}$$

Now, a single sentence vector is computed for both questions by aggregating the vectors h_i and the weights α_i with summation as follows,

$$v = \sum_i \alpha_i h_i$$

7. Predict / Classify - This involves the final MLP with sigmoid activation as follows,

   $\widehat{y} = sigmoid(W_c v + b_c)$

   the predicted class is given by $\widehat{y} = argmax_i(\widehat{y_i})$

The negative log-likelihood loss for the model is given by,

$$L = -\sum_d log\, \widehat{y_d}$$

The learnable parameters of the model are:

W_w , b_w , u_w of the Word Attention Layer

W_s , b_s , u_s of the Sentence Attention Layer

The hyperparameters of the model are:

1. Dropout regularization rate for the GRU for the Word Encoder
2. Dropout regularization rate for the GRU for the Sentence Encoder
3. Optimization method (Adam / RMSProp / Nadam / SGD)
4. Learning rate for the optimization method
5. Number of samples (batch size) per gradient update
6. The numbers of epochs used for training

Prediction

The following is the classification report,

	precision	**recall**	**f1-score**	**support**
not duplicate	0.88	0.84	0.86	84267
duplicate	0.74	0.80	0.77	49148
micro avg	0.82	0.82	0.82	133415
macro avg	0.81	0.82	0.81	133415
weighted avg	0.83	0.82	0.83	133415

Model Accuracy Score

The accuracy, recall and prediction scores for the model were computed as follows:

accuracy score	0.8237
precision score	0.7401
recall score	0.8038

Conclusion

It is obvious from the model evaluation results for the various approaches used, that XGBoost as well as Deep Learning with Hierarchical Attention are effective ways to solve the problem of duplicate identification. Though the accuracy for the deep learning methods are less than that for XGBoost, it is to be kept in mind that the XGBoost model was tuned whereas the deep learning models were not. Further tuning of the deep learning models, would most probably improve the accuracy of those models.

Further Exploration

Exploration of hyperparameter tuning for the deep learning models using Hyperas could improve the accuracy of these models.

References

NLP Resources

1. Spacy
2. Gensim
3. Fasttext
4. FuzzyWuzzy

Models

1. Logistic Regression
2. XGBoost
3. Hyper-parameter Tuning
4. Embed, encode, attend, predict: The new deep learning formula for state-of-the-art NLP models, by Matthew Honnibal
5. Hierarchical Attention Networks for Document Classification by Yang et al.
6. A Decomposable Attention Model for Natural Language Inference by Parikh et al.

Deep Learning Resources

1. Keras
2. TensorFlow
3. Hyperas