CSC_51054_EP

Data Challenge Report: Sub-Event Detection

Weight Watchers

Jonas Berger / Victor Micha / Kenneth Browder

jonasberger38@gmail.com / michavictor@gmail.com / pandanautinspace@gmail.com

<u>Section 1: Data Preprocessing and Feature Selection/Extraction</u>

1. Data preprocessing

We begin by reading all the .csv files from the training set and putting the data in a single data frame, using the *pandas* library. The timestamp column is dropped for convenience, as we assume that the order of the tweets is retained in the PeriodIDs. We then rename the MatchIDs and IDs to have match ids in increasing order from 0,1,2… for ease of debugging and convenience. This is also done for the evaluation tweets, and we just have to keep track of the actual match IDs by using a simple dictionary. This is very easy to do and only requires 2-3 extra lines of code while having the benefit of easier debugging and testing.

For the preprocessing of the text on tweets, we apply the following operations: lowercase, removal of punctuation/numbers/stopwords, and lemmatization (reducing each word to its base form or lemma). While lemmatization changes the words, it does not change their overall meaning, so the overall syntactic message is intact.

For the tweet embeddings, we use the "glove-twitter-200" model, which is trained specifically on tweets and converts tweets into 200-dimensional vector embeddings. For every tweet, we get the vector embedding of each word and take the overall mean. Essentially, every tweet is converted to one 200-dimensional vector, which is the mean of each embedded word. This is a common and simple way of ensuring that the overall syntactic meaning of the tweet is preserved. Once we have the embeddings, we can drop the Tweet column and add the 200 columns corresponding to the tweet embeddings.

In order to deal with the many tweets that have the same MatchID and PeriodID, we must aggregate them to create a vector that is easy to feed to the LSTM. We do this by grouping the data by MatchID and PeriodID, then generating an average embedding vector for each group, leaving no duplicate PeriodID rows per match, while also significantly decreasing the size of the data. We also drop the ID column now since it encodes the same information as the MatchID and PeriodID columns. To ensure that the PeriodIDs remain in chronological order and are grouped by their MatchID, we convert these columns to integers and sort the rows by these values.

We now have the final data format, with columns MatchID, PeriodID, EventType, and 200 columns for the tweet_vector, where the MatchIDs are ordered from 0 to 15 and the PeriodIDs are ordered chronologically. This is crucial since we will take advantage of this sequentiality by using an LSTM model as our main model.

The first 80% of matches (first 13 matches) of the train tweets are used for training, and the last 20% (last 3 matches) are used for testing. Now that we have the training and test data, we format the data to be input for the PyTorch LSTM model. We are using batch_first=True for convenience, where the input 3D tensor must be of the shape (batch_size, seq_len, num_features), or in our case (match_id, period_id, num_features=200). We are treating each match as 1 sequence, a sequence of periods where each period has 200 features, which is the tweet embedding for that period as described above. The usage of the tensors are "X_tensor[match_id][period_id]", which returns a list of length 200 of corresponding tweet vector, and "y_tensor[match_id][period_id]", which returns the corresponding EventType (1 or 0).

Since not every match has the same number of periods, we pad the shorter sequences with 0s. This will be dealt with by using 'pack_padded_sequence' and 'pad_packed_sequence' before and after passing the data through the LSTM, respectively. We also keep track of the total number of periods for each match/sequence in an array. This ensures that the LSTM only processes the actual sequence content, ignoring padded values, which prevents the model from learning patterns from padding (noise).

Typically for LSTM, scaling must be done to ensure that the values are within a certain range [1], typically in the ranges of [0, 1] or [-1, 1]. Since the outputs of GloVe-twitter-200 are already scaled between [-1,1], there is no need to apply further scaling [2].

2. Motivations behind the features

We used *GloVe-twitter-200* for tweet embeddings, as it is a set of pre-trained word embeddings trained on a large corpus of tweets. According to a study, pre-trained embeddings like *GloVe-twitter-200* outperformed traditional methods, which suggests that the embeddings we used are very effective for sentiment analysis of tweets [3]. The embeddings are intended to capture the semantic meaning and sentiment of the tweet and give valuable information on the following: if the tweet captures a lot of emotions, it might be more likely that a significant event happened during this period. However, if the tweet is rather bland, such as a tweet simply commenting how boring the match is, and conveys little to no strong emotion, it might be more likely that no significant event happened during this period.

For the LSTM, we made sure to order the tweet embeddings chronologically for each match, effectively exploiting the order of the tweets as much as possible. Similar to other recent work on sub-event detection for Twitter [8][9], our reasoning for believing that the order of the tweets are of great importance is that in football games, significant events that occur earlier in the game can have a drastic effect on the game later on. For example, a red card, goal, injury, or other significant event can give a significant advantage or disadvantage to one team which might only be manifested later on in the game.

3. Comparison of various combinations of features

With the LSTM as the model, we experimented with different methods to construct features in order to determine the optimal balance between complexity and effectiveness. We experimented at multiple stages during the feature creation process: generating embeddings / vectorization, sequence construction, and aggregation. For each combination of features, we attempted to keep the rest of the pipeline (preprocessing, training of the model) as uniform as possible to isolate the feature construction as the factor to compare. Results of the testing are shown in Table 2.

For the vectorization stage of feature collection, we tried two ways of representing tweets as vectors for input to LSTM, the GloVe-twitter-200 method described above, and a most-common n-gram approach. In the n-gram approach, we found the 200 most common n-grams (we used n=8, so most country names would fit in one n-gram, and 200-most-common to give a vector comparable to GloVe) in the whole training corpus, and counted the occurrence of the n-grams from in the tweets in each period. This approach with long n-grams puts it between n-gram frequency and word frequency, two approaches commonly used as baselines in event/sub-event detection [10][11]. Since every period had a different size, these vectors were normalized to have a magnitude of 1. Compared to GloVe, performance was no better with n-grams, and was usually worse.

During the sequence construction phase, along with the match sequences described above, we also constructed sequences of vectors for each period, based on splitting the period into a number of “chunks” based on the timestamp. We experimented with feeding the periods directly to the LSTM, modifying the architecture of the LSTM slightly to only use the output of the last hidden layer, as well as constructing a longer sequence for each match by simply concatenating the period sequences together, and then using a number of methods for sampling the correct number of output layers to feed forward. We also experimented with varying chunk sizes for both of these period-sequence based approaches. The long-sequence match performed better and trained more quickly than the direct period method, but even with the best sampling method (striding over the hidden layers, sampling after every period), it brought no gains over the original match sequence approach.

Finally, we experimented with different methods of aggregation from vectors per-tweet, to vectors for every period. The original method was to take the mean of all the tweet vectors to generate a single vector of length 200, but we also experimented with augmenting the mean vector by concatenating the standard deviation of these tweet vectors as well, to give the model a sense of the distribution of the vectors. The model performed very well with these augmented vectors, achieving the highest maximum accuracy on the test set, but performed poorly on the evaluation set. Thus, choosing the mean-pooling form of aggregation puts us in alignment with similar LSTM-based approaches [8].

Section 2: Model Choice, Tuning and Comparison

For the training, we tried three categories of models: KNN, Decision Trees (DT), and finally Long Short-Term Memory (LSTM). The performance of the various models on the test and evaluation sets are listed in Table 1 at the end of the report.

1. First models and observations

KNN gives a first idea of the results the dataset can give. The parameters are already normalized values in a latent space, so their distance could be significant. It shows, however, its limitations due mostly to the high dimensionality of the dataset, and possibly a clustering that is not significant enough. The model was also tested on lower dimensions using PCA, but it showed no improvement. To better demonstrate this issue, we computed the PCA decomposition and made a visualization of the first dimensions, i.e. the two eigenvectors with the highest eigenvalues (cf Figure 1).

We can't see any meaningful separation between the data with and without events. As a comparison, we also computed the PCA decomposition on a single match (cf Figure 2). This time the separation is clearer, showing a distinction of the features of the match, which means that models like KNN would probably perform better if trained on a single match, but also that the patterns of the features are different between each match. This is a problem, especially since our model must perform on new, unknown matches, and needs to be able to generalize patterns.

Next, to handle the high-dimensionality of the dataset, we tried to use Decision Trees. This category of model is efficient on large datasets and naturally prioritizes important attributes, which is good for our embedded features as we cannot naturally interpret their importance. However, the basic Decision Tree model generally causes overfitting, suffers from non-robustness and is really sensitive to randomness. To minimize those issues, we improved the model to Bagging Trees and Random Forest. Those two minimize the variance of DT and have a better control on overfitting.

We trained those 3 models using the *scikit-learn* library and a grid-search cross-validation algorithm. As expected, the Decision Tree made the worst accuracy and the Random Forest gave encouraging results.

We then trained two Gradient-Boosting Tree algorithms, namely XGBoost [4] and LightGBM [5]. Those algorithms are more complex and thus less intelligible, but are known to be both adaptive and resistant to overfitting. We once again tuned them using a grid-search algorithm. These two methods gave better results than Random Forest. Their main "disadvantage" is that they are not adapted to NLP, and thus do not take advantage of the properties of word embeddings, such as the sequentiality of words in sentences. This is logically why we focused our work on the LSTM model.

2. LTSM model

For the LSTM, the task of hyperparameter tuning to maximize the test accuracy was done automatically, by coding a function that, given a list of possible hyperparameters, returns the parameters with the highest accuracy on the test data. We only consider accuracy on the test data because this is the primary evaluation metric for this project. Since the prediction accuracy over multiple runs for the same parameters varies, we take the average over multiple runs. However, because of the randomness of dropout, running this function multiple times does sometimes give different answers. To prevent overfitting, we use a dropout rate of 0.2, which is good practice for certain LSTM models according to a study [6]. For the training of the LSTM, we noticed that the loss was relatively high, so to tackle this we used a mask to compute the loss only on non-padded elements. The mask has shape [num_sequences, max_seq_length], with 1s for valid positions and 0s for padded positions in each sequence. Using this technique, we can set the loss of the padded values to zero and finally normalize the loss by the number of valid (non-padded) elements. In order to perform this loss optimization technique, we set reduction = 'none' for the BCELoss: `nn.BCELoss(reduction='none')`.

For the final predictions, we convert probabilities obtained from the sigmoid function to 0 or 1 using 0.5 as a threshold. However, since we notice that most correct answers for the evaluation tweets are 1, we experimented with a lower threshold such as 0.4, which would increase the chance of classifying an output as 1. We found a research paper that supports this claim, explaining that thresholds other than 0.5 for unbalanced data can be beneficial. However, we did not notice a significant difference [7].

References:

[1] AP Monitor. *“LSTM Network”*. March 3rd, 2022 https://apmonitor.com/do/index.php/Main/LSTMNetwork

[2] Wolfram Neural Net Repository. *“GloVe 200-Dimensional Word Vectors Trained on Tweets”*. June 21st, 2018 https://resources.wolframcloud.com/NeuralNetRepository/resources/GloVe-200-Dimensional-Word-Vectors-Trained-on-Tweets/

[3] Elmitwalli, S., Mehegan, J. *“Sentiment analysis of COP9-related tweets: a comparative study of pre-trained models and traditional techniques”*. March 20th, 2024 https://pmc.ncbi.nlm.nih.gov/articles/PMC10987730/

[4] Chen, T., Guestrin, C. *“XGBoost: A Scalable Tree Boosting System”*. June 10th, 2016 https://arxiv.org/pdf/1603.02754

[5] Ke, G. et al. *"LightGBM: A Highly Efficient Gradient Boosting Decision Tree”*. December 4th, 2017) https://papers.nips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[6] Tanjina, C. et al., *“A regularized LSTM method for detecting fake news articles”*. November 16th, 2024 https://arxiv.org/html/2411.10713v1

[7] Brownlee, J. (2021) *“A gentle introduction to threshold-moving for Imbalanced Classification, MachineLearningMastery.com”*. January 5th, 2021 https://machinelearningmastery.com/threshold-moving-for-imbalanced-classification/

[8] Giannis Bekoulis, Johannes Deleu, Thomas Demeester, and Chris Develder. 2019. Sub-event detection from twitter streams as a sequence labeling problem. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 745–750, Minneapolis, Minnesota.

[9] Nguyen VQ, Anh TN, Yang H-J. Real-time event detection using recurrent neural network in social sensors. International Journal of Distributed Sensor Networks. 2019;15(6). doi:10.1177/1550147719856492

[10] Meladianos, P., Nikolentzos, G., Rousseau, F., Stavrakas, Y. and Vazirgiannis, M. 2021. Degeneracy-Based Real-Time Sub-Event Detection in Twitter Stream. Proceedings of the International AAAI Conference on Web and Social Media. 9, 1 (Aug. 2021), 248-257. DOI:https://doi.org/10.1609/icwsm.v9i1.14597

[11] A. Dhiman and D. Toshniwal, "An Approximate Model for Event Detection From Twitter Data," in IEEE Access, vol. 8, pp. 122168-122184, 2020, doi: 10.1109/ACCESS.2020.3007004

Appendix

Model	Hyperparameters	Accuracy on test set	Accuracy on eval set
KNN	Nb neighbors: 3	58.00%	62.89%
Decision Tree	Min samples leaf: 10 Min samples split: 25	46.46%	-
Bagging	*(Optimized Decision Tree model)* Nb estimators: 14	58.46%	-
Random Forest	Max feature: sqrt Min samples leaf: 2 Nb estimators: 40	66.92%	61.72%
XGB	Nb estimators: 100 Max depth: 1 Learning rate: 0.15	**68.77%**	67.58
LGBM	Nb estimators: 100 Max depth: 4 Learning rate: 0.05	68.46%	68.36%
LSTM	hidden size: 32, num layers: 3 dropout rate: 0.2 num epochs: 400 learning rate: 0.005	64.10%	**70.31%**

Table 1: Comparison of the different models' accuracies with certain hyperparameters

Feature		**Highest Accuracy Achieved on Test Set**	**Accuracy on Eval Set**
Vectorization	Most common n-grams	70.98 %	X*
	GloVe	75.xx %	**70.31 %**
Sequence Representatio n	Period	73.01 %	69.92 %
	Match	75.xx %	**70.31 %**
	Period+Match	73.89 %	66.80 %
Aggregation Method	Mean	75.xx %	**70.31 %**
	Mean & Standard Deviation	**78.xx %**	64.83 %

Table 2: Comparison of Various Feature Combinations

*: Did not test most common n-grams on Evaluation set due to low accuracy on test set.

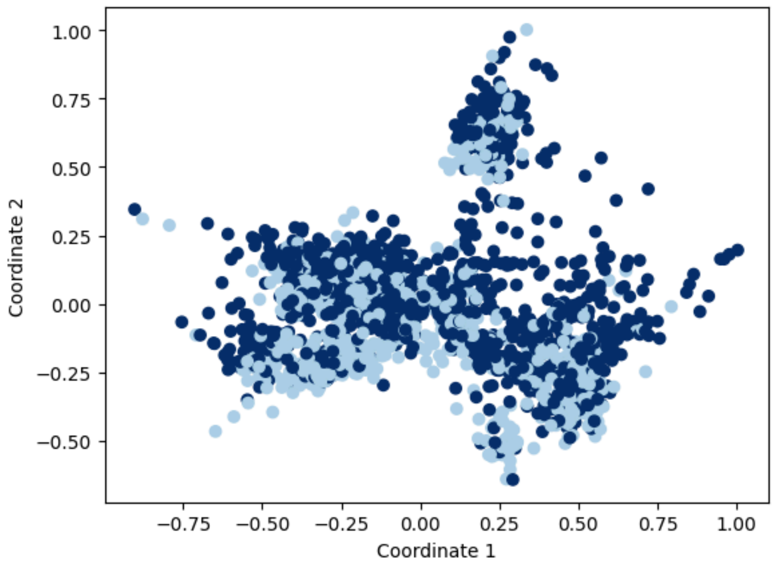

Figure 1: PCA applied on the whole dataset

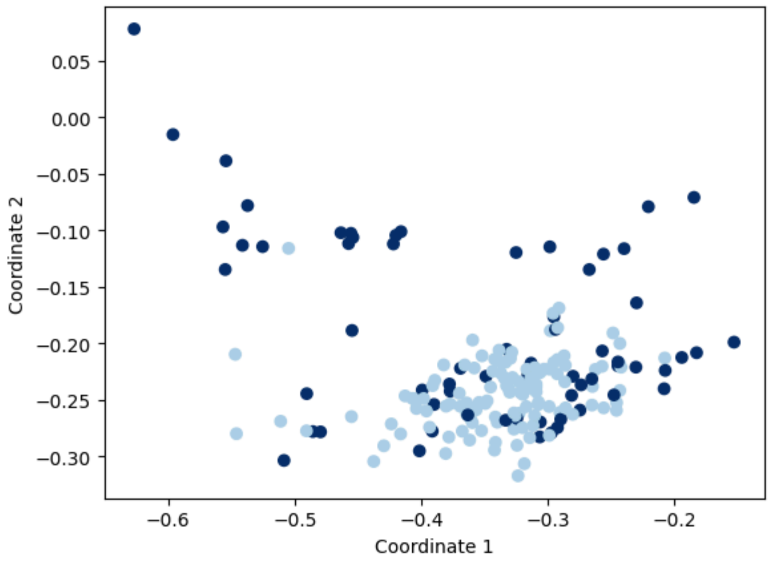

Figure 2: PCA applied on the dataset corresponding to the 10th match (GermanyBrazil74)