COMPGI19 Assignment 2 Report

Kapileshwar Syamsundar

Coauthor
Affiliation
Address
email

1 Implementation of Perceptron Algorithm

For this problem, we implemented a Perceptron trainer with the default setup; the results of our trainer and the precompiled trainer are shown in Table 1.

Table 1: Classification performance for perceptron implementations

	Our perceptron trainer	Precompiled perceptron trainer
Average precision	0.5128644939965694	0.5262206148282098
Average recall	0.6842105263157895	0.665903890160183
Average F1	0.5862745098039216	0.5878787878787879

The results show that our trainer has lower precision and higher recall than the precompiled trainer but, overall, has roughly the same harmonic mean of the two (F1).

2 Implementation of Average Perceptron

Our first implementation of the averaged perceptron involved keeping a running sum of the weights used for each prediction and then, after each instance and iteration, dividing the weights by the number of predictions made. However, this nave implementation is slow because every weight (even those that have not changed) is added to the sum of weights after every prediction and, in this domain, there are potentially many hundreds of thousands of features and respective weights. Therefore accessing and adding all of these weights for every training instance of every iteration is very inefficient.

Instead, we devised an alternate algorithm for accumulating the weights used for each prediction during training: for each prediction, only weights that are being changed (i.e. those associated with a candidate and the gold or predicted label) are added to the sum of weights, drastically reducing the number of weights accessed after each prediction. We achieve this by storing the last time each weight was modified so that when a weight is about to be changed, the duration of the current weight is known and can be multiplied by the weight to give the sum of that weight over the duration; this can then be added to the running sum of weights.

The results for our averaged perceptron with the default setup show it to be a lot better than the precompiled perceptron; these are shown in Table 2.

Table 2: Classification performance for average perceptron implementations

	Our average perceptron trainer	Precompiled average perceptron trainer
Average precision	0.453125	0.21044045676998369
Average recall	0.5308924485125858	0.5903890160183066
Average F1	0.48893572181243417	0.31028262176788934

3 Feature Engineering and Evaluation

In this section we outline and motivate potential feature templates for trigger and argument classification. Where possible, concrete training set examples are also provided. The best set of features (for both perceptron and Naive Bayes learning methods) are then presented.

Naive Bayes is a generative model and perceptron is discriminative. Generative models account for the prior probability $p(x)$ of seeing a token. Maximising this model results in maximising $\log(p(y|x)) + \log(p(x))$. In text classification, we only care about finding the most likely label for a token, i.e. only maximising $p(y|x)$ and not $p(x)$. By optimising for both, we may sacrifice higher values for $p(y|x)$ to optimise for $p(x)$, giving us less accurate label predictions. However in the perceptron model, we only optimise for $p(y|x)$. This is why we expect the perceptron learning algorithm to perform better.

In choosing features for Naive Bayes, we consider that features are all independent and are weighted equally. This means that features should be tailored towards identifying specific labels instead of discriminating against the token being of a certain label. Further, choosing features that are similar will end up in maximising the prior probability instead of $p(y|x)$, resulting in poorer classification ability. We choose a subset of distinct features from the features outlined below that showed the best discriminative power.

Papers that provided inspiration for features have been referenced next to the feature name.

3.1 Trigger features

3.1.1 Lexical

Stem of word. Many events of the same label share a common stem. The transcription event is commonly associated with the token 'transcription', for instance. More generally in cases where we want to use the text of the token, we take the stem instead, allowing us to directly compare the meaning of the token. The suffix that we drop is an indication of its POS tag which we are given.

Our highest weighted feature for this class is having the stem ?transcript' for Transcription events. The feature also works well to discriminate against labels. For example, when the stem is ?inhibit?, a low weighting is given to the token being Positive regulation.

POS of word. [1] We expect words like conjunctions to be less likely to indicate events; typically we expect events to be verbs since they indicate an action. The highest weighted feature in this class is for tokens with POS tag CC (coordinating conjunction) to have label None. The lowest weighted feature is for tokens with POS tag VBN (verb, past participle) having label None, confirming our intuition that events tend to be verbs.

We see that this feature performs strongest in determining whether or not a token is of label None.

Capitalisation. [1] This feature counts the number of capital letters in the token. The immediate downside of this feature is that we also consider tokens that start at the beginning of a sentence which are capitalised regardless of their event classification.

3.1.2 Entity-based

Proteins in sentence. We count the number of proteins in the sentence. Here our intuition is that having no proteins in the sentence makes it less likely that the sentence contains any event. We see that our highest weighted feature for this feature template is assigning a label of None to the current token if there are no proteins in the sentence giving evidence that our intuition is correct. However, this feature template doesn't discriminate well between events of type other than None.

3.1.3 Syntax-based

Dependencies. [4] This class of features looks at the grammatical relations between words. Dependencies are an edge between two related words 'head' and 'mod' where 'head' refers to the source of the dependency and 'mod' is dependent on head. Experimentally we see that only considering dependencies as a feature doesn't perform well (this is a feature we implemented but chose not to

select in the final set), so we also consider the POS and stem of the related dependency to our token. We think this is because whatever the explanation is.

We consider features involving the mod/head, stem of the parent, dependency and POS. Here our token is the head. We consider these dependencies up to a depth of 3, meaning that if we have an edge between tokens A and B, we also consider the dependency between B and C and C and D (if they exist). An example of a well performing feature is assigning the Binding label to tokens where the edge to its associated token is prepAs (prepositional binding) and the associated token's POS is VBZ (verb, third person singular present).

3.1.4 Other

Context. [1] We capture the stems of immediately surrounding tokens. These tokens give us an understanding of the context in which the token is being used. We see in many cases that the previous token gives a clear indication of what the current token is. Our highest weighted feature for this class highly weights tokens being labelled as Transcription when the previous token is 'mRNA'. The previous word helps limit the events that the current token can be, so it works well in discriminating between all event labels. As a result, we see that we have highly negatively weighted features as well such as assigning the label Gene expression when the previous token is 'mRNA'.

Features which look at the next token do not discriminate between the current token's possible labels as well as the previous token, but they help nonetheless. An example is assigning the label Positive regulation when the next token has stem 'chromosom'.

3.1.5 Perceptron features

• Stem of word • POS of word • Number of capital letters in word • Prior word • Next word • Number of proteins in sentence • Level 1 dependency count • Level 2 dependency count • Level 3 dependency count • Level 1 dependency type and POS of mod • Level 2 dependency type and POS of mod • Level 3 dependency type and POS of mod • Level 1 dependency type and stem of mod • Level 2 dependency type and stem of mod • Level 3 dependency type and stem of mod • Level 1 dependency type and stem of head • Level 1 dependency type and POS of head

3.1.6 Naive-Bayes features

• Stem of word • Number of capital letters in word • Number of proteins in sentence • Level 1 dependency type and POS of mod

3.2 Argument features

3.2.1 Lexical

POS. Part-of-speech features can provide semantic and contextual information related to the candidate and parent tokens of the data sets. For example, theme arguments generally have a part-of-speech value of 'NN', which can be used to differentiate theme arguments from cause and none types. Looking at the part-of-speech value for events can reveal information related to their child arguments as certain events are likely to have a specific part-of-speech related to them. The part-of-speech values can be combined to create unique pairs of them which can be used to map a specific argument classes to part-of-speech pairings in a one-to-many relationship.

Value of word. The word value of candidates and parents can be used as base measures to classify arguments as they will give a base relation between specific words and how they are related to arguments. Certain argument types are likely to relate to specific sets of words so creating features based on these values should provide a large amount of data for the classifier models to use. Conjunctions of word based features can also be used to create specific word pairs or sets that can be used to target argument types with higher precision.

Stem. Stem features provide a more generalised version of word features as the they group words related to a common theme. These can be very useful for determining what event the parent of the candidate relates to and in turn can reveal a significant amount of information related to the argument class of the candidate.

Capitalised letters in candidate. [2] As certain candidates in the training set have capital letters in their names, we can use the set of capitalised letters to identify proteins more accurately and apply the information to classifying arguments.

3.2.2 Entity-based

Candidate is a protein. Upon inspection, many of the arguments of type cause and theme contain the protein tag under the mentions section. Therefore, we can filter out the arguments of type none by identifying whether the candidate is a protein or not.

Number of proteins. Following on from the last feature, if there are no proteins in a sentence, then it is likely that the arguments in the sentence are of type none. As such, counting the number of proteins in the sentence can reveal the classification of the candidate.

3.2.3 Syntax-based

McClosky dependency between candidate and parent. [1] In general, a specific type of McClosky dependency exists between an event and its arguments. Therefore, we can use these dependencies to map to specific argument types in order to classify a candidate.

Number of McClosky dependencies on candidate. Counting the number of dependencies on the candidate can provide insight on what argument type it may be as arguments of a specific type can have more dependencies than others. In addition to this, it also recognises events which are also arguments effectively as they are likely to have many dependencies and relationships with other tokens in the data.

Number of McClosky dependencies on parent. Counting the number of dependencies on the parent can help classify which type of event it may relate to and therefore assist in classifying the candidate to a specific argument type.

3.2.4 Other

Prior word. As described in trigger features.

Prior stem. Creates a more general case for the prior word comparison by grouping words related to a specific theme.

Prior POS. For the same reason as the other prior techniques, creating pairs of parts-of-speech provide more contextual data to the classification process.

Absolute token distance between candidate and parent. [3] From observing the training data, it can be seen that arguments are often located close to their parent token, meaning that bounds can be set to filter out arguments of type none. This value can be obtained as an absolute value or signed value to determine whether the candidate lies on the left or right side of it?s parent which can be used to obtain more positional information for specific arguments. Upon inspection of the training data, most distances above the value of 40 tended to be none-type arguments so the value was used as a cutoff point feature to classify none tokens.

Candidate part-of-speech is 'NN' and is a protein. A combination of part-of-speech and whether the candidate is a protein or not can be used to classify theme arguments specifically. From observation of the training set, there are an abundance of cases where a theme argument has the part-of-speech 'NN' and is also a protein.

3.2.5 Perceptron features

• Boolean check part-of-speech of candidate and parent are equivalent • Combination of the part-s-of-speech of the candidate and parent • Combination of the stem of the candidate and parent • Combination of stem of parent and candidate protein tag check • Combination of if a capital letter exists in the candidate and protein tag check • Number of proteins in the sentence • Candidate protein tag check • Dependency from candidate to parent • Dependency from parent to candidate • Stem of prior word and candidate • Part-of-speech of prior word and candidate • Absolute distance between the candidate and parent in token units

3.2.6 Naive-Bayes features

• Combination of candidate part-of-speech and protein tag check • Boolean check if candidate and parent have equivalent parts-of-speech • Boolean check if a capital letter exists in the candidate and if it has a protein tag • McClosky dependency between the candidate and it?s parent • Boolean check if absolute distance between candidate and parent is less than 40 • Boolean check if the number of proteins in the sentence is greater than zero • Prior part-of-speech check for candidate

3.3 Results

Table 3: Trigger classification results

Learning algorithm	Feature set used	Average precision	Average recall	Average F1
Perceptron	Perceptron	0.1940	0.8132	0.3133
Naive Bayes	Perceptron	0.0142	0.1323	0.0256
Naive Bayes	Naive Bayes	0.1609	0.7500	0.2650

First we implemented lexical features, yielding an average F1 score of 0.1237. Adding entity features increased the score to 0.1735. Syntax features increased the score by a considerable amount to 0.2540. Finally including all other features raised the score to 0.3133.

Table 4: Argument classification results

Learning algorithm	Feature set used	Average precision	Average recall	Average F1
Perceptron	Perceptron	0.07338	0.8675	0.1353
Naive Bayes	Perceptron	0.01311	0.9066	0.02585
Naive	Naive Bayes	0.06565	0.6635	0.1195

As additional features were applied to the models, the improvements to the scores were recorded to provide insight as to which groups of features provided the largest overall benefit. For the perceptron model, the lexical features alone provided an average F1 score of 0.0375 showing that alone they did not provide accurate classifications. However, adding entity features improves the score to 0.0494 showing that these features work well in conjunction with each other. Adding syntactic features using the McClosky dependencies between words improves the score further by to 0.0628 and adding prior word features on top that pushes the score to 0.0985. Finally, adding positional features provides the final score of 0.1353.

For the Naive Bayes model, using only the lexical features provides an F1 score of 0.12 but it does not classify any cause statements whatsoever. Therefore, additional features were added in an attempt to classify cause at the expense of the F1 score. Adding entity features didn?t provide any assistance in classifying cause and also didn?t alter the F1 score. However, adding syntactic and positional features allowed for the model to classify theme arguments and only changed the F1 score to 0.119.

4 Joint Perceptron

4.1 Unconstrained joint model

The joint classifier classifies arguments and triggers together. This means that the output of the classifier's prediction function is a tuple that contains the best choice of labels (as dictated by the weights from training) for the trigger and its arguments. This can be represented as a sum of the score for the event trigger and the scores for its arguments. As the score for the triggers and arguments is dependent on their labels, maximising the overall score corresponds to finding the best labels.

Because the unconstrained model is such that the label assigned to a trigger is independent to the labels assigned to its arguments, maximising the total score is equivalent to individually maximising the trigger and argument scores. So given a candidate, the relevant feature vector, weights and a set of possible labels for the candidate, we can define a generic argmax routine to return the most likely label. The pseudo code for such an argmax algorithm is shown below:

```
argmax(labels, candidate, weights, feat):
  scores = []
  for (elem in labels):
   featureVector = feat(candidate, elem) // create feature Vector
   score = dot(weights, featureVector)
   scores.append((score, elem))

   // return the label of the score with highest score value
   return maxBy(scores.elem).label
```

4.2 Constrained joint model

As with the unconstrained version of the joint model, the scores are calculated for each trigger label and then for the argument labels for each argument and these are summed to give the overall scores for the joint structure.

For the constrained version of the joint model, the score for each trigger label is calculated and stored, rather than just storing the score of the label that maximises the score for the trigger. Then, for each trigger label, the argument labels are predicted with the required constraints in place.

Two of the constraints, 'A trigger can only have arguments if its own label is not NONE' and 'Only regulation events can have CAUSE arguments', are enforced by removing illegal argument labels from the set of possible argument labels. For example, when the trigger label is 'None', the arguments labels 'Theme' and 'Cause' are removed from the set of legal argument labels. These legal argument labels are the ones used in calculate the argmax of the arguments.

The final constraint, 'A trigger with a label other than NONE must have at least one THEME', is implemented by, for each argument, calculating the score of it being 'Theme' plus the maximum score of the other arguments given the available labels. The maximum score of these is then added to the score of the trigger and the trigger and argument labels that give the maximum of these trigger-argument combined scores is returned as the argmax for this joint constrained model.

This method of implementing the constraints is a lot more efficient than calculating all possible combinations of arguments and their scores for each label and then filtering out those that violate the constraints before returning the label combination with the maximum score.

```
argmax(triggerLabels, argLabels, x):

  for (tlabel in triggerLabels):
    triggerScore(tlabel) = score(x, tlabel)
    if (tlabel is "None") {
      currentArgLabels = argLabels - "Theme" - "Cause"
    }
    else if (tlabel is not a regulation event label) {
      currentArgLabels = argLabels - "Cause"
    }
    else {
      currentArgLabels = argLabels
    }

  for (arg in x.arguments):
    argScore(arg) = score(arg, "Theme") + ...
        sum(maxScore(x.arguments != arg))
    totalScore(tlabel) = triggerScore(tlabel) + max(argScore)
    return max(totalScore).labels
```

5 Implementation and Evaluation for Problem 4

Using the default split of 80% training data and 20% development data from 500 documents, we trained the per-task models from Problem 3 and the two joint models from Problem 4 over the default number of iterations, 10.

Table 5: Performance of models

Candidate	Metric	Per-task Models	Unconstrained Joint Model	Constrained Joint Model
Trigger	Average precision	0.1940	0.1806	0.2729
	Average recall	0.8132	0.8003	0.7691
	Average F1	0.3133	0.2947	0.4029
Argument	Average precision	0.07338	0.1362	0.1864
	Average recall	0.8675	0.6330	0.3880
	Average F1	0.1353	0.2242	0.2518

The average F1 results in Table 5 show that the joint models are a lot better at predicting the argument labels than the per-task model, with the constrained joint model being the best. Although the recall of argument labels is decreased for both joint models, the precision is increased, resulting in a higher average harmonic mean between the two metrics. However, it is the opposite for trigger classification: the joint models perform worse than the per-task model, with the constrained joint model performing the worst. The unconstrained joint model does have a higher recall than the per-task model for triggers but the decrease in precision for both models results in a lower average harmonic mean. It is likely that the reason for the constrained joint model having the highest average precision for arguments is that the set of labels is reduced for most argument candidates as a result of the constraints.

6 Error Analysis

6.1 Best model

Overall we found our best model to be the joint constrained model, trained on the perceptron learning algorithm. The performance metrics we valued the highest when choosing our best model were precision and harmonic mean. While recall scores indicated a measure of completeness (quantity), we opted for precision?s measure of exactness (quality) as a more relevant performance indicator.

6.2 Types of errors

With an average precision of 0.273, all labels suffered from errors in classification. The event label with the least misclassifications was phosphorylation while the model struggled with precision for regulation events the most. It was found that the most common types of errors (across all types of triggers) were mislabelling candidate triggers to be none. The following sentence shows an instance of this.

?Activation and expression of the nuclear factors of activated T cells, NFATp and NFATc, in human natural killer cells: regulation upon CD16 ligand binding.?

For argument classification, average precision was 0.187. As with trigger classification, it was found that the main error was misclassifying candidate arguments to be of type none. An example instance of when this happened is shown below.

?Both nuclear run-on and actinomycin D pulse experiments strongly indicate that HU regulates c-jun mRNA expression by increasing the rate of synthesis as well as stabilizing the c-jun mRNA.?

The frequency of these types of errors can be inferred from tables 6 and 7, where the columns represent the predicted label and the rows represent the true label.

Table 6: Predictions vs true labels for event triggers

Gold — Prediction	Phosphorylation	Negative regulation	Regulation	Protein catabolism	Binding	Positive regulation	Localization	Transcription	None	Gene expression
Phosphorylation	18	0	0	0	0	0	0	0	2	0
Negative regulation	0	47	3	0	0	6	0	1	10	5
Regulation	0	0	63	0	0	11	1	0	13	3
Protein catabolism	0	0	0	10	0	0	0	0	5	0
Binding	0	1	1	0	54	6	0	0	11	7
Positive regulation	0	1	10	0	0	172	0	0	32	13
Localization	0	0	0	0	0	0	25	0	0	6
Transcription	0	0	1	0	0	0	1	38	5	13
None	12	83	279	11	174	564	31	89	5649	241
Gene expression	0	0	0	0	0	5	1	0	4	166

Table 7: Predictions vs true labels for argument triggers

Gold — Prediction	None	Theme	Cause
None	84070	1763	3
Theme	554	407	0
Cause	77	11	0

6.3 Errors it helps to prevent

In the joint models, arguments and triggers are predicted together and so classification occurs together. This idea is very useful for argument extraction as an argument is defined over multiple words (source trigger and the argument candidate itself).

So the types of error that joint models help prevent are mainly applicable to arguments where prior knowledge of the trigger label influences the labels for its corresponding arguments. For example, if it is known that the trigger is regulation, then the arguments for such a trigger can be limited to only be of label cause.

As a result, even if the trained weights predict an illegal label, through constraints we can force classifications to a label (potentially the correct one) of a lower score.

6.4 Unresolved errors

Using joint model, we can resolve further errors with argument features by enforcing further constraints and heuristics from studying the dataset that target specific trigger labels and limit the range of legal possibilities for given argument candidates. This would primarily help resolve errors where candidates with true labels that are not none are mislabelled as other non ?none? labels. However, as observed by the error analysis, these instances are quite rare.

Although the constraints lower the number of possible label combinations to predict, it does not address the problem of candidates misclassified as none. This problem lies with the features chosen. The current features do not help to strongly discriminate between identifying candidates with label none against other specific labels. This is why the majority of the incorrectly predicted candidates are labelled as none. In the case of the label Cause, it is always predicted as none Engineering features that capture information more relevant to specific labels (as opposed to those that are more generic to any label) are how such errors can be resolved.

Another way to potentially resolve errors could be to change from a joint model to a pipeline one. In this pipeline model, triggers could be predicted labels first and then these predictions incorporated when predicting arguments. This means feature templates for arguments could access predicted parent token labels. The pipeline model would still effectively be a joint model but the constraints are implemented on a lower level instead, where they can be of a more specific and complex nature.

Appendices

A Examples of trigger features

B Examples of argument features

References

[1] Lishuang Li, Yiwen Wang, and Degen Huang. Improving feature-based biomedical event extraction system by integrating argument information. *Proceedings of the BioNLP Shared Task 2013 Workshop*, page 109115, 2013.

[2] MAKOTO MIWA, RUNE SAETRE, JIN-DONG KIM, and JUNICHI TSUJII. Event extraction with complex event classification using rich features. *Journal of Bioinformatics and Computational Biology*, page 135, 2010.

[3] Arzucan Ozgur and Dragomir R. Radev. Supervised classification for extracting biomedical events. *Proceedings of the Workshop on BioNLP*, page 111114, 2009.

[4] Deepak Venugopal, Chen Chen, Vibhav Gogate, and Vincent Ng. Relieving the computational bottleneck: Joint inference for event extraction with high-dimensional features. . *In Proceedings of EMNLP*, 2014.