COMPGI19 Assignment 2 Report

Reece Doyle **Navid Hallajian** **Kapileshwar Syamasundar** **Oliver Tun**

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	0.5129	0.5262
Average recall	0.6842	0.6659
Average F1	0.5863	0.5879

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, dividing the weights by the number of predictions made. However, this naive implementation is slow because every weight is added to the sum of weights after every prediction. In this domain, there are potentially many thousands of features and weights; summing all of these weights for every training instance and iteration is inefficient.

Instead, we devised an alternate algorithm for accumulating the weights during training: for each prediction, only weights that are being changed 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 and, 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	0.4531	0.2104
Average recall	0.5309	0.5904
Average F1	0.4889	0.3103

3 Feature Engineering and Evaluation

We first consider the difference between the perceptron and Naive Bayes learning algorithms. Generative models like Naive Bayes 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 $p(y|x)$. Optimising for both, sacrifices higher values of $p(y|x)$ to optimise for $p(x)$, giving less accurate predictions. However, the perceptron model only optimises for $p(y|x)$ and so it is expected for this model to perform better.

For Naive Bayes, features are independent and weighted equally. This means that features should be tailored towards identifying specific labels instead of discriminating against the tokens of a certain label. Further, choosing similar features results in maximising the prior probability leading to poorer classification. We choose a distinct subset showing the best discriminative power from the features below.

We outline and motivate potential feature templates for trigger and argument classification. Where possible, evidence is provided either through training set examples or highly weighted features from training. The best set of features (for both perceptron and Naive Bayes methods) are then presented. Papers that provided inspiration for features have been referenced next to the feature name.

3.1 Trigger features

3.1.1 Lexical

Stem of token. Many events of the same label share a common stem instead of the word itself; to generalise better, we use the stem. This feature 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 token. [1] Words like conjunctions are less likely to indicate events; we expect events to be verbs since they indicate an action. The highest weighted feature in this class was for tokens with POS *CC* (coordinating conjunction) to have label *None*, supporting our claim. 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. In addition to capturing when a candidate was the first word of a sentence, without the need for creating an extra feature template, we can also infer if the candidate was a protein.

3.1.2 Entity-based

Number of proteins in sentence. Having no proteins in the sentence makes it less likely that the sentence contains any event. Our highest weighted feature for this template was assigning a label of *None* to the current token if there were no proteins in the sentence, giving evidence that our intuition is correct. However, this feature template doesn't discriminate well between non-*None* events.

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*. Experimentally only considering dependencies as a feature didn't perform well, so we also consider the POS and stem of the related dependency to our token to give more context.

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). An example is given in Figure 1.

3.1.4 Other

Context. [1] We capture the stems of immediately surrounding tokens. They give us a better understanding of the context in which the token is being used. In many cases, the previous token gives a clear indication of what the current token is. Our highest weighted feature for this class labels tokens 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*.

Looking at the next token did not discriminate between labels as well as looking at the previous token, but it helped. An example found in the training set was 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. These features provide semantic and contextual information related to the candidate and parent tokens. For example, *Theme* arguments generally have a POS value of *NN*, differentiating *Theme* from *Cause* and *None* labels. POS values for events reveal information related to their child arguments as certain events are likely to have a specific POS related to them. Figure 2 shows an example of classifying *Theme*.

Value of word. Certain argument types relate to specific sets of words so creating features based on these values helps capture this information.

Stem. Stem features provide a more generalised version of word features as the they group words related to a common theme. These can be useful to determine what event trigger the parent token relates to, in turn revealing information related to the argument label of the candidate.

Capitalised letters in candidate. [2] Since some candidates have capital letters in their name, we can use the set of capitalised letters to identify proteins more accurately.

3.2.2 Entity-based

Candidate is a protein. Many of the arguments of type *Cause* and *Theme* are proteins. We can filter out arguments of type *None* by identifying if the candidate is a protein. Figure 3 demonstrates this.

Number of proteins. Following on from the last feature, if there are no proteins in a sentence, it is likely that the arguments in the sentence are of type *None*. Counting the number of proteins in the sentence can capture this information.

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. We can use these dependencies to map to specific argument types in order to classify a candidate. Figure 4 demonstrates this.

Number of McClosky dependencies on candidate. Counting the number of dependencies on the candidate can provide insight on its argument type Different argument types may have more dependencies than others. Additionally, it also recognises events that are also arguments since they are likely to have many dependencies and relationships with other tokens.

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

Context. This feature template refers to using the prior token as described in trigger features, including considering its stem (as a more general case of using the full prior word) and its POS value to capture sentence context.

Absolute token distance between candidate and parent. [3] We observed that arguments are often located close to their parent token, meaning that bounds can be set to filter out arguments of type *None*. Upon further inspection, most distances above the value of 40 tended to be arguments of type *None*. As a result, the value was used as a cut-off point to classify *None* tokens.

Candidate POS is *NN* and is a protein. From observation, there were many cases where a *Theme* argument had the POS value *NN* and was also a protein. Figure 5 demonstrates this.

3.2.5 Perceptron features

• POS of candidate and parent are equivalent • POS of candidate and parent • Stem of candidate and parent • Stem of parent and candidate is protein • Candidate is protein and has capital letter • Number of proteins in sentence • Candidate is protein • Dependency from candidate to parent • Dependency from parent to candidate • Stem of prior word and candidate • POS of prior word and candidate • Absolute distance (number of tokens) between candidate and parent

3.2.6 Naive-Bayes features

• POS of candidate and candidate is protein • POS of candidate and parent are equivalent • Candidate is protein and has capital letter • McClosky dependency between candidate and parent • Absolute distance (number of tokens) between candidate and parent • Number of proteins in sentence > 0 • Prior POS of 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

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

During trigger classification, when training on the perceptron we first implemented lexical features, yielding an average F1 score of 0.1237. Adding entity features increased the score to 0.1735. Addition of syntax features increased the score to 0.2540 and other features raised the score to 0.3133. As shown in Table 3, using these features on the Naive Bayes model significantly lowered the score and so a separate feature set was engineered for this.

During argument classification, when training with the perceptron lexical features alone yielded an F1 score of 0.0375. Adding entity features improved the score to 0.0494 and addition of syntactic features raised the score to 0.0628. Finally, adding other features raised the score to 0.1353. As with triggers, a separate set of features were needed when running on the Naive Bayes model. Results for these are shown in Table 4.

From these results we found that in all performance aspects, the perceptron model was the best learning algorithm for both classification tasks (as expected).

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 pseudocode for such an argmax algorithm is shown below:

```
argmax(labels, candidate, weights, feat):
  for (label in labels):
    featureVector = feat(candidate, label) // create feature Vector
    score = dot(weights, featureVector)
    scores(label) = score
  return max(scores).key // return highest-scoring 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. 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 to calculate the argmax of the arguments.

For each argument, the final constraint *"A trigger with a label other than None must have at least one Theme"* is implemented by 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 summed with 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) // independent trigger score
    if(tlabel is "None"): // apply constraints 1 and 3
      currentArgLabels = argLabels - "Theme" - Cause
    else if(tlabel is not a regulation event label):
      currentArgLabels = argLabels - "Cause"
    else:
      currentArgLabels = argLabels
    if (tlabel is not None):
      for (arg in x.arguments): // apply constraint 2, one arg must be Theme
        argScore(arg) = score(arg,Theme) + sum of max score of other arguments
```

```
    else:
      argScore(arg) = sum of max scores of all arguments
  totalScore(tlabel) = triggerScore(tlabel) + max(argScore)
  return labels that gave max(totalScore)
```

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 results in Table 5 show that, overall, the joint models improve upon the per-task models. The unconstrained joint model is a massive improvement over the isolated argument model: with higher average precision and a sacrifice of average recall resulting in a higher harmonic mean (F1) of the two metrics. However, the unconstrained joint model performs worse than the isolated trigger model in both precision and recall, but the improved argument labelling certainly outweighs this. The constrained model further improves upon the F1 score for arguments despite a decrease in average recall. It is likely that the precision of the constrained model for arguments is greater than that of the unconstrained model because the constraints restrict the number of labels that are available, given the labelling of the parent trigger. This decreases the probability of an argument being mislabelled.

For trigger classification, the isolated model performs slightly better than the unconstrained model but the constrained model has a much higher average F1 than both. As with the arguments, the constrained model has lower average recall than the other two models for triggers but the average precision is much better. Again, this is likely to be due to the constraints restricting the labels available for the arguments, based on the labelling of the trigger. This means that if the most likely label for a trigger is incorrect and its arguments violate the constraints for that label, then it is likely that the trigger will be relabelled according to what fits best for the labelling of the arguments.

6 Error Analysis

6.1 Best performing 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 Error prevention ability

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).

The types of error that joint models help prevent are mainly applicable to arguments where prior knowledge of the parent trigger's label influences the labels for these arguments. E.g., if it is known that the trigger is *None*, then the arguments for such a trigger can be limited to only be of label *None*. Therefore, correct prediction of a label for one candidate will aid in the correct prediction of labels for all other candidates in the same structure.

6.4 Unresolved errors

Using the constrained 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*. The current features do not help to strongly discriminate between candidates with label *None* and 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 to a pipeline model. In this pipeline model, trigger labels could be predicted first, then used 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, where they can be of a more specific and complex nature.

7 Joint Conditional Likelihood

The conditional log-likelihood objective can be expressed as follows.

$$\begin{aligned}
\text{CLL} &= \sum_{x_i,c_i} \log p_\lambda(c_i|x_i) \\
&= \sum_{x_i,c_i} \log(Z^{-1}_{\lambda,x_i}) \exp S_\lambda(c_i,x_i) \text{ where } Z_{\lambda,x_i} \text{is the normalizer} \\
&= \sum_{x_i,c_i} S_\lambda(c_i,x_i) - \log Z_{\lambda,x_i} \\
&= \sum_{x_i,c_i} S_\lambda(c_i,x_i) - \log \sum_{c_i^{'}} \exp S_\lambda(c_i^{'},x_i)
\end{aligned}$$

The normaliser can also be expressed as the maximum score over all $c_i^{'}$ for a training example:

$$\begin{aligned}
&= \sum_{x_i,c_i} S_\lambda(c_i,x_i) - \log \max_{c_i^{'}}(\exp) S_\lambda(c_i^{'},x_i) \\
&= \sum_{x_i,c_i} S_\lambda(c_i,x_i) - S_\lambda(\hat{c_i},x_i) \text{ , where } \hat{c_i} = \arg\max_{c_i^{'}} S_\lambda(c_i^{'},x_i) \\
&= \sum_{x_i,c_i} \sum_j \lambda_j f_j(c_i,x_i) - \sum_j \lambda_j f_j(\hat{c_i},x_i) \text{ , since } S_\lambda(c_i,x_i) = \sum_i \lambda_i f_i(c_i,x_i)
\end{aligned}$$

Therefore, the partial derivative is

$$\frac{\delta \text{ CLL}}{\delta f_j} = \sum_{x_i,c_i} f_j(c_i,x_i) - f_j(\hat{c_i},x_i)$$

Giving the gradient

$$\sum_j \sum_{x_i,c_i} f_j(c_i,x_i) - f_j(\hat{c_i},x_i)$$

As the derivation shows, the gradient of the conditional likelihood objective is the empirical expectation minus the predicted expectation for each feature and training example. During each epoch of gradient ascent, the gradient needs to be evaluated but this can take $\mathcal{O}(nd)$ time, where n is the number of training examples and d is their dimensionality. It is therefore prohibitively expensive to calculate these expectations for every epoch: this is a naive approach.

A tractable method for evaluating the gradient would involve either not using all training examples or not using all features; our proposed method chooses the former of these two. Instead of summing the expectations of all training instances, a training instance is uniformly chosen at random to be used in calculating the expectations and then this is multiplied by the size of the training set, n, to give the gradient. For a training instance (c,x), we have $n \sum_j f_j(c,x) - f_j(\hat{c},x)$.

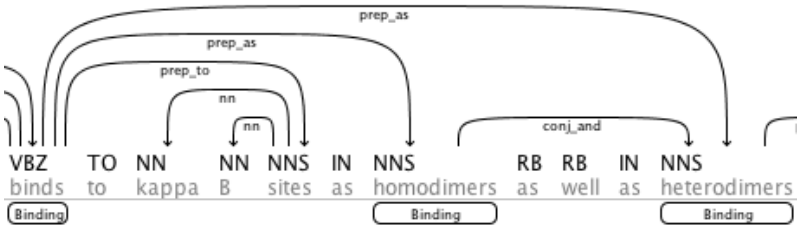
The pseudocode for the algorithm is given below.

```
gradient(trainingSet, features):
  sum = 0
  instance = random instance from trainingSet // (label,data) tuple
  for(feat in features):
    predicted = argmax(instance) // returns the predicted label for the instance
    sum += feat(instance) - feat(predicted,instance.data)
  return sum * trainingSet.size
```

Appendices

A Examples of trigger features

Figure 1: Dependency type of POS of mod



B Examples of argument features

Figure 2: POS of candidate and parent

Figure 3: Candidate is a protein

Figure 4: McClosky dependency between candidate and parent

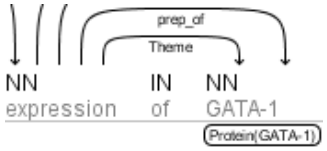


Figure 5: Candidate POS is NN and is a protein

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.