Hillel Mendelson 049840143
Nadav Yutal 066031121

29.1.2017

# NLP Wet 2

## Training

### Feature representation

We decided to go with an approach similar to the one taken in wet 1, because of the “feature family” nature of features, instead of using sparse matrices, for the $f \cdot v$ calculation, we decided to preserve feature family structures. These structures use a hash to represent exact features which hold the feature index and count of occurrences on the parsed corpus and feature index to hash.
This representation is very easy to work with, to calculate a feature vector for a given edge on the dependency graph, one only needs to “ask” each feature generator if the derived hash from the sentence and the edge words exists.
For example, to get the feature vector for a given sentence and edge:

```
    def get_features_for_edge(self, sentence, head, counter):
        res = np.zeros(self.get_size())
        for fg in self.fgArr:
            k = fg.get_feature_idx(sentence, head, counter)
            if k != -1:
                res[k] += 1.
        return res
```

### Chu-Liu Edmonds

We took the chu-liu edmonds code from the github link provided in the homework sheet. There was little modification necessary for it, to return maximum spanning tree and not minimum spanning tree (multiply the weights by -1 as input to the algorithm).
Another modification needed for it was to convert from our Sentence class to graph dictionary as needed for the algorithm.
<u>Drawbacks of this algorithm:</u>

- When using this algorithm for finding y’ is that features can’t include dependency features which include more edges other than the current edge (for example head of head, siblings etc..). This limitation occurs due to the fact that the structure of the graph is dynamic (at first full clique, at the end the mst).
- Because the supplied code is a general mst generator, there were cases where the mst included <u>multiple edges from the root to other nodes</u>. We will mention how we solved it in the basic model section.
- Finally, we suspect the supplied code is a little bit buggy, there were cases that the mst generated by the code didn’t bring back a graph with all nodes. (we ignore the errors but this is clearly not correct)

Hillel Mendelson 049840143
Nadav Yutal 066031121 29.1.2017

## Parallel Perceptron

We noticed that the perceptron takes a very long time to run: 5000 sentences X mst X N iterations.
We had a few ideas about expediting the algorithm:

1. Run the mst concurrently from all nodes, dropping the head node.
   Pros: This would yield more accurate results, as the current algorithm allows for multiple arcs from the head which is not a legal representation graph.
   Cons: this would create a possible slowdown on a machine with less cores than the average number of words in a sentence. $O(n^3) \rightarrow O(n^4)$ / num_of_cores
2. We decided to go with a parallel perceptron. This idea is documented in:
   http://www.cslu.ogi.edu/~bedricks/courses/cs506-pslc/articles/week3/dpercep.pdf
   A simple divide and conquer approach is not enough as the resulting weights, trained on a smaller corpus would not be “magically” mixed into a correct weight vector.
   One needs to average the weights of each returned worker after each epoch to keep the gradient in the right direction.This is somewhat similar to batch processing in deep networks.
   PerceptronIterParamMix:

```
a. Shard T into S pieces T = {T1, . . . , Ts}
b. w = 0
c. for n : 1..N
    i. w(i,n) = OneEpochPerceptron(Ti , w)
    ii. w = sum(μ*w(i,n))    (μ=1/n)
d. return w
```

3. Modular training: we’ve built the model so we could incrementally add new families, without damaging the data structure. This way, if one has previous weights from a long training, they could be reused. Simply add the new family at the end, and run a few more perceptron rounds to readjust the weights.

## Filtering

We decided to simplify the filtering process and control it using one parameter that affects all the feature generators. E.g. set this parameter to 3 will filter each feature that didn’t show up more than 2 times.
We know that this method isn't perfect, but it effectively removes the majority of sparse features.

Hillel Mendelson 049840143 29.1.2017
Nadav Yutal 066031121

## Basic model - first iteration

Our first model included families 1-13. These produced 397,151 features as follow:

```
{'F7': (71232, 'filter=0'), 'F11': (70679, 'filter=0'), 'F1': (9993, 'filter=0'), 'F13':
(749, 'filter=0'), 'F9': (70401, 'filter=0'), 'F10': (33936, 'filter=0'), 'F12': (69819,
'filter=0'), 'F3': (37, 'filter=0'), 'F2': (8876, 'filter=0'), 'F4': (15908, 'filter=0'),
'F5': (14162, 'filter=0'), 'F6': (45, 'filter=0'), 'F8': (31314, 'filter=0')}
```

## Basic model - weaker root

We wanted to compensate for the mst returning multiple arcs from the root.
In this model, each feature generator that includes header information is dropped hence doesn't get score for this feature.
Total feature count was 394,032, not a dramatic change, and there was no dramatic changes in the score.

## Basic model - filtering

We knew that filtering will not be very helpful in the basic model because it's used to prevent overfitting which wasn't our problem in this model. However, it was interesting to see how filtering will affect the results. We chose to run training with filter=2
The number of features after filtering the weak root model was 108,700, less than ⅓ from the original set, and as we'll show below, essential information was lost from these sparse features:

```
{'F13': (624, 'filter=2'), 'F4': (7347, 'filter=2'), 'F3': (35, 'filter=2'), 'F5': (6886,
'filter=2'), 'F7': (13693, 'filter=2'), 'F2': (6990, 'filter=2'), 'F9': (14219, 'filter=2'),
'F1': (7789, 'filter=2'), 'F8': (10255, 'filter=2'), 'F6': (45, 'filter=2'), 'F10': (12582,
'filter=2'), 'F12': (14392, 'filter=2'), 'F11': (13843, 'filter=2')}
```

## Complex model

The full model included much more features: 1,247,263. These included:
- Absolute distance between header and modifier
- Direction from head to modifier (left, right)
- Steps from head to modifier, including negative values to represent left/right
- CROSS(Previous features X the modifier part-of-speech)
- Contextual features as described in Michael Collins' lecture. neighbours of modifiers/head information such part of speech or words, and some crosses between them.

Hillel Mendelson 049840143 29.1.2017
Nadav Yutal 066031121

- In-between features as described in Michael Collins lectures. Include part of speech up to distance of 15 words between head and modifier and up to 8 words which include cross of all of part of speech.

In our first training we decided to add a global filter of 2. The features and their count is as follows:

```
fv contains  365367  features after filtering
{'FSentenceLengthWithPos': (2173, 'filter=2'), 'FInBetween8': (1022, 'filter=2'),
'FModifierNeighLR1': (7664, 'filter=2'), 'FInBetween4': (1892, 'filter=2'), 'FDirection':
(2, 'filter=2'), 'F11Direction': (13695, 'filter=2'), 'F2': (6990, 'filter=2'),
'FHeadNeighLR6': (1000, 'filter=2'), 'FHeadNeighLR3': (18079, 'filter=2'), 'FInBetween5':
(1630, 'filter=2'), 'FDist': (101, 'filter=2'), 'FDirectionWithPos': (45, 'filter=2'),
'FInBetween6': (1420, 'filter=2'), 'FInBetween12': (622, 'filter=2'), 'FModifierNeighLR6':
(1114, 'filter=2'), 'FInBetween1': (2654, 'filter=2'), 'F10': (12582, 'filter=2'),
'FModifierNeighLR2': (4979, 'filter=2'), 'F10Direction': (12492, 'filter=2'), 'F7': (13693,
'filter=2'), 'FHeadNeighLR7': (587, 'filter=2'), 'FInBetween9': (896, 'filter=2'), 'F11':
(13843, 'filter=2'), 'FHeadNeighLR4': (19423, 'filter=2'), 'F9Direction': (14093,
'filter=2'), 'F1': (7789, 'filter=2'), 'FInBetween7': (1222, 'filter=2'), 'F7Direction':
(13562, 'filter=2'), 'FModifierNeighLR7': (883, 'filter=2'), 'F12Direction': (14249,
'filter=2'), 'FInBetween10': (785, 'filter=2'), 'F4': (7347, 'filter=2'), 'FAbsDist': (61,
'filter=2'), 'FHeadNeighLR2': (3558, 'filter=2'), 'FModifierNeighLR8': (859, 'filter=2'),
'FModifierNeighLR5': (11688, 'filter=2'), 'F13Direction': (876, 'filter=2'), 'FInBetween3':
(2273, 'filter=2'), 'FHeadNeighLR1': (26417, 'filter=2'), 'FDistWithPos': (1010,
'filter=2'), 'F9': (14219, 'filter=2'), 'F3': (35, 'filter=2'), 'F13': (624, 'filter=2'),
'FMissedBigrams1': (12520, 'filter=2'), 'F5': (6886, 'filter=2'), 'FSentenceLength': (82,
'filter=2'), 'FModifierNeighLR3': (11237, 'filter=2'), 'FMissedBigrams2': (10278,
'filter=2'), 'FAbsDistWithPos': (690, 'filter=2'), 'FHeadNeighLR8': (560, 'filter=2'), 'F6':
(45, 'filter=2'), 'FModifierNeighLR4': (11728, 'filter=2'), 'F8Direction': (10314,
'filter=2'), 'F8': (10255, 'filter=2'), 'FInBetween11': (680, 'filter=2'), 'F12': (14392,
'filter=2'), 'FInBetween2': (2630, 'filter=2'), 'FInBetween13': (532, 'filter=2'),
'FHeadNeighLR5': (12390, 'filter=2')}
```

A day before the submission date we decide to try the full model, which eventually produced the best results. It included the features as follow:

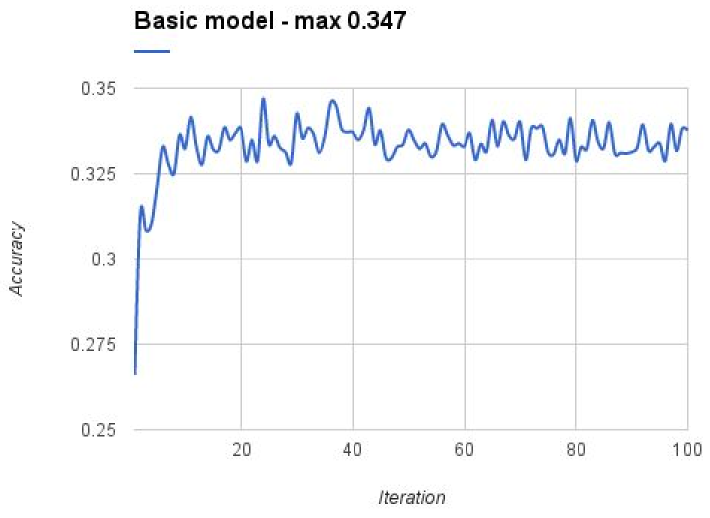
```
fv contains  1247263  features before filtering
{'FMissedBigrams2': (29104, 'filter=0'), 'FInBetween7': (2558, 'filter=0'), 'F2': (8876,
'filter=0'), 'F3': (36, 'filter=0'), 'F8Direction': (31903, 'filter=0'), 'FSentenceLength':
(82, 'filter=0'), 'FInBetween4': (3727, 'filter=0'), 'F12Direction': (70818, 'filter=0'),
'F6': (45, 'filter=0'), 'FInBetween6': (2915, 'filter=0'), 'F11Direction': (70614,
'filter=0'), 'FInBetween5': (3294, 'filter=0'), 'FHeadNeighLR8': (620, 'filter=0'),
'F7Direction': (71077, 'filter=0'), 'FModifierNeighLR7': (1044, 'filter=0'), 'FInBetween10':
(1821, 'filter=0'), 'FInBetween13': (1338, 'filter=0'), 'F11': (69678, 'filter=0'),
'FAbsDistWithPos': (867, 'filter=0'), 'F9Direction': (71344, 'filter=0'), 'FHeadNeighLR6':
(1096, 'filter=0'), 'FMissedBigrams1': (32218, 'filter=0'), 'FModifierNeighLR1': (95053,
'filter=0'), 'F13Direction': (1079, 'filter=0'), 'F10': (33918, 'filter=0'),
'FModifierNeighLR5': (50728, 'filter=0'), 'FHeadNeighLR7': (647, 'filter=0'),
'FInBetween12': (1481, 'filter=0'), 'FModifierNeighLR6': (1302, 'filter=0'), 'FAbsDist':
(75, 'filter=0'), 'FInBetween9': (2047, 'filter=0'), 'FHeadNeighLR1': (52440, 'filter=0'),
'F13': (731, 'filter=0'), 'F9': (70401, 'filter=0'), 'FInBetween8': (2304, 'filter=0'),
'FInBetween3': (4242, 'filter=0'), 'FDistWithPos': (1328, 'filter=0'), 'FDist': (125,
'filter=0'), 'FModifierNeighLR3': (48650, 'filter=0'), 'FDirectionWithPos': (45,
'filter=0'), 'FInBetween1': (4573, 'filter=0'), 'F1': (9992, 'filter=0'), 'F4': (15908,
'filter=0'), 'FDirection': (2, 'filter=0'), 'FModifierNeighLR4': (49681, 'filter=0'),
'FHeadNeighLR3': (30053, 'filter=0'), 'F5': (14162, 'filter=0'), 'F10Direction': (36191,
'filter=0'), 'F12': (69819, 'filter=0'), 'FHeadNeighLR4': (29737, 'filter=0'), 'F7': (70192,
'filter=0'), 'FHeadNeighLR2': (4455, 'filter=0'), 'FInBetween11': (1617, 'filter=0'),
'FModifierNeighLR8': (1017, 'filter=0'), 'FInBetween2': (4720, 'filter=0'), 'F8': (30274,
'filter=0'), 'FHeadNeighLR5': (22779, 'filter=0'), 'FModifierNeighLR2': (7937, 'filter=0'),
'FSentenceLengthWithPos': (2483, 'filter=0')}
```

# Inference

We decided to save the weights after each iteration in order to get a deeper view on how our models behave. For our final model each training iteration took ~14 minutes. Around 23 hours for 100 iterations.
Inference was performed by generating full clique graphs and calculating each edge’s weight. Then sending it to the edmonds algorithm which returns an mst that represents our predicted dependencies.
The final stage includes passing over all the sentences and for each word and its header in the sentence check if exists a corresponding edge in our predicted dependency graph.

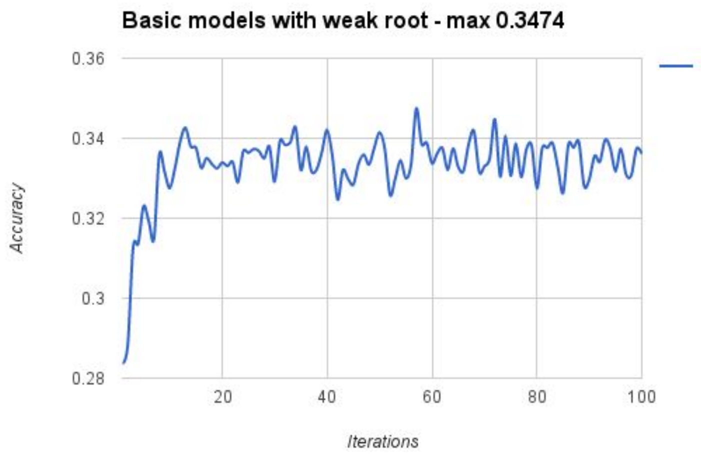
## Basic model

The model performed poorly on the test data. It quickly converged (10 iterations) to ~33% and did not further improve. The reason for this is that the given features were too simple to catch a lot of the tree structure nuances.



## Basic model with weak root

Also as mentioned earlier, the supplied edmonds algorithm allows the root to include more than 1 edge, hence we got some mst trees with multiple edges going out from the root. We first tried to give all the weights going from root a 0 score in order to add it only at the last iteration, but for some reason the edmonds implementation didn’t like it and broke with some error message on missing stack elements. Because we had only a short time to handle this project we decided to give weak scores for the root edges, and that was the trigger for the weak root connection model.



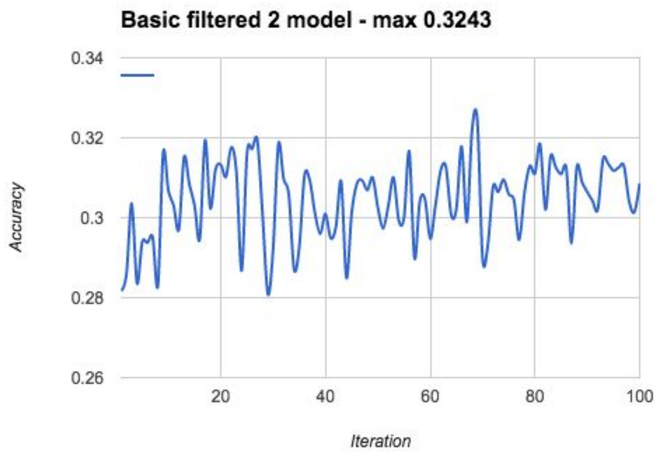
We did see a better maximum accuracy, but still the performance is very low.
Our scores per requested iterations:

| **20** | **50** | **80** | **100** |
|---|---|---|---|
| 0.3339362795 | 0.3415005139 | 0.3274820144 | 0.3361151079 |

Although best score achieved at iteration 57, We choose iteration 100 from that model to represent m1 because the mdoel look more stable on that ares (but still very noisy)

## Basic model with filter=2

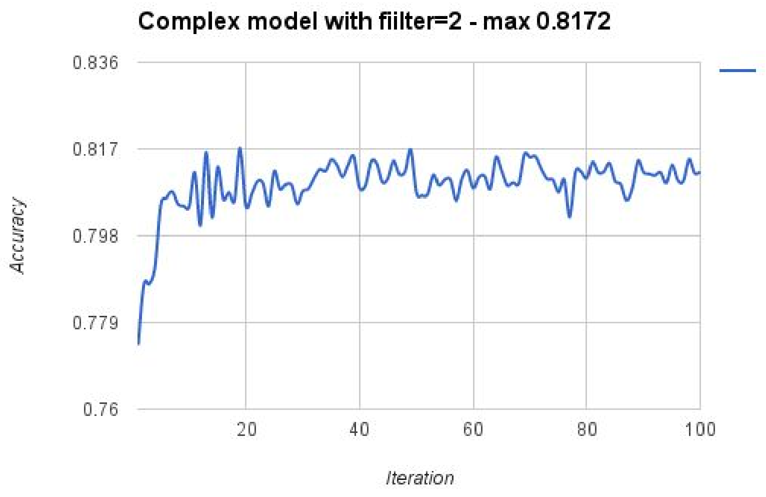
As expected and explained in the training section, the results were worse. Although the slightly worse accuracy, it’s main advantage was its small feature count (less than ⅓).



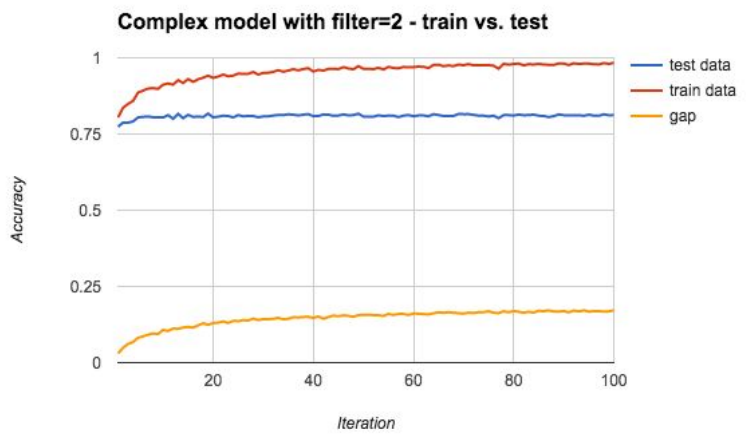
Hillel Mendelson 049840143 29.1.2017
Nadav Yutal 066031121

## Complex model

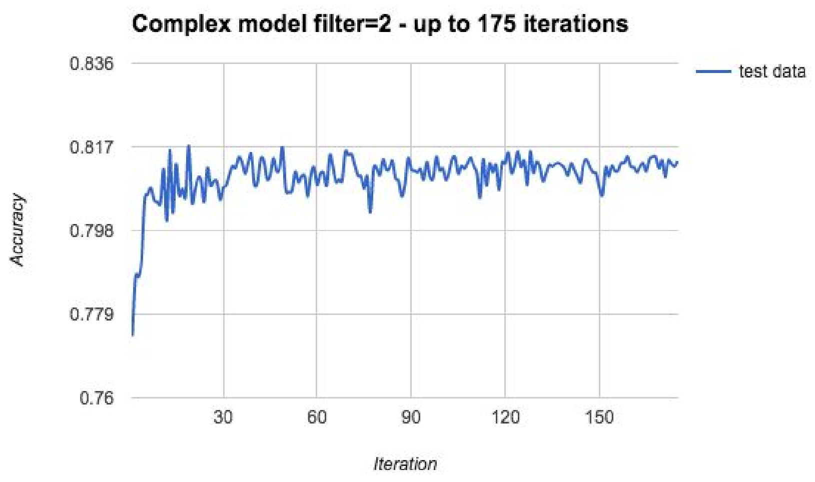
We first ran the filter=2 model and got a maximum accuracy of 0.8172. This result was achieved on iteration 19 where the perceptron still toggles a lot.
We felt that the weights still slightly improve over time, and that perhaps iteration 19 would not be a good choice for other corpuses. So we decided to further explore)



Next, we wanted to see how well this model performs on the training data and produced a graph that contains both of test and training and also the gap between them:

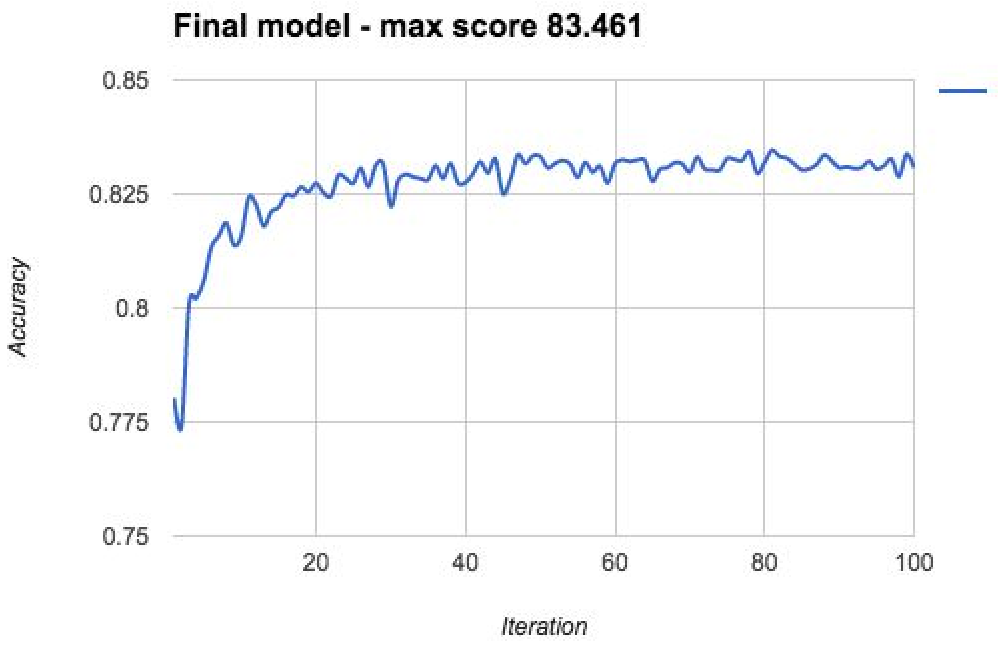


It seems like around iteration 20 we cross the “shoulder” where the gap almost stops growing. If so, maybe this model isn’t expressive enough too.
Even though we knew we would not gain accuracy from more iterations, we wanted to see if this model was going to be more stable so we added up to 175 iterations.
Below you can see that indeed it looks more stable as the number of iterations increased. If we should pick weights from this model (we didn’t..) we would go with the 175th iteration that produced “only” 81.385% accuracy but resides in a much stable zone.



## Final model - bigger is better

With little more than 24 hours to go, we decided to try an unfiltered model (over a million features) and did not regret it:



This model outperform all other models and got stable around 0.831 accuracy after enough iterations. The peak accuracy 0.83461 occurred on iteration 81. And we decided to take those iteration weights for the competition, although we don't think it really matter what to take from that area.

Hillel Mendelson 049840143 29.1.2017
Nadav Yutal 066031121

# Competition

In order to generate the competition files we did the following steps:

- Parse the file with very similar code of the train/test parsing with one little change that fills head=-1 for each word.
- Generate a feature vector from the participated features (same as in test)
- Load specific chosen weights for each model (same as in test)
- For each sentence generate a full clique and calculate its edges weights
- Produce an mst graph which represents the predicted dependency tree by sending the clique to the edmonds algorithm
- Update words with their predicted head
- Print to file the updated corpus

## Reproduce competition result:

1. In Open conf.py file:
    a. Change is_competition to True
    b. Change weights_src_comp to desired weights file
       For m1: m1.weights
       For m2: *m2.weights*
    c. Change is_complex (True if m2, False if m1)
    d. Change comp_output_file_name to desired name
2. Run dependency_parser with python 3
3. Enjoy!!!

As we got it hard on the first competition, We predict lower results than the test accuracy. That for several reasons:

1) We optimize the model against test data and not competition data.
2) The data of the competition may arrived from different topic than the train/test