UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Drug-Drug Interaction Detection and Classification

Master in Data Science, FIB
Mining Unstructured Data

April 18th, 2023

Pau Comas Herrera - Odysseas Kyparissis

Table of Contents

Introduction

The definition of drug-drug interaction (DDI) is broadly described as a change in the effects of one drug by the presence of another drug. The detection of DDIs is an important research area in patient safety since these interactions can become very dangerous and increase healthcare costs. Medical literature and journals are powerful sources for Information Extraction about those interactions, a process that aims to automatically extract the interaction knowledge without the need for our healthcare professionals to constantly review the literature.

Information Extraction can be addressed as a Natural Language Processing task, in which Relation Extraction focuses on extracting possible relations between previously detected entities present in a text. In our case, having the entities as the drugs and the relations as the interactions give us a methodology to address the challenge. With the DDI Corpus, which is a semantically annotated corpus of documents describing drug-drug interactions from the DrugBank[1] database and MedLine[2] abstracts, we face the goal of detecting when there is an interaction and when not, and classifying the existing drug-drug interactions into ***advice***, ***effect***, ***mechanism***, and ***int***. We'll explain their meaning in a further section. The key objective of the project is therefore to learn about the effect of different features and models for this problem.

In this report we present which algorithms we used and the rationale behind that, how we addressed the feature extraction challenge for them and what were the results of our experimentation and testing. Finally, the report concludes with final remarks and insights gained from the task.

Machine Learning DDI

Selected Algorithms

We experimented with two ML algorithms, Multinomial Naive Bayes and Linear Support Vector Machine (Linear SVM).

The Naive Bayes (NB) classifier is a probabilistic classifier that works on the principle of Bayes' theorem. It assumes that all features are independent of each other and uses this assumption to calculate the probability of a given feature belonging to a particular class. The Multinomial variant of NB, which was used as one of the two classifiers, is specifically designed for working with discrete features, such as word counts or frequency of occurrence. We chose this classifier because it is simple to implement and has a low computational cost. In that way, it was possible to compare feature configurations, their complexity differences and derive potential conclusions about the tradeoff between predictability power and overhead.

[1] https://go.drugbank.com/
[2] https://www.nlm.nih.gov/medline/medline_overview.html

On the other hand, SVMs are known for their high accuracy and effectiveness in handling complex data with large feature spaces. A Linear SVM is a type of SVM that uses a linear kernel to separate the data points. It is linear because the decision boundary that it generates is a straight line that separates the data into different classes. This makes it suitable for linearly separable datasets. However, if the dataset is not linearly separable, then a linear SVM may not perform well. We took this risk given that our demands for the algorithms were that we wanted them to be simple, parametrizable, and fast to use. The focus is put on the feature extraction process and its effects on the models predictability more than achieving the optimal complex model.

Feature Extraction

In this chapter, we will discuss the different features that were tried, discarded, and used in the development of a best-performing feature extraction function.

The input to the feature extractor is not the sentence itself, but a dependency tree with added individual features about every node/word, generated with CoreNLP[3]. CoreNLP (Core Natural Language Processing) is a popular open-source software library developed by Stanford University that provides a suite of natural language processing tools for a variety of tasks, including tokenization, part-of-speech (POS) tagging, named entity recognition, parsing, sentiment analysis, and coreference resolution. It is written in Java and can be used as a standalone tool or integrated into other applications, such as ours.

In a dependency tree, each word is a node in the tree, and the arcs between the nodes represent the grammatical relationships between the words. For example, a typical dependency tree will have a root node representing the main verb of the sentence, with arcs pointing to its subject and object. The CoreNLP dependency tree object contained, for every word/node in a sentence, a dictionary with the following relevant information:

- Literal word
- Lemma of the word
- Part-of-speech (POS) tagging
- Address of head node
- Relations/dependencies with other nodes
- Offsets in the sentence

And then from there, with the help of the *deptree.py* module we could also obtain or infer:

- Least Common Subsumer (LCS) between words
- Paths in the dependency tree (labeled as we wanted)
- Head entity of a node
- Who is drug entity

[3] https://stanfordnlp.github.io/CoreNLP/

Original set of features

We were provided with a template with some features, which applied in the adult Multinomial NB resulted in a macroaverage F1 of 35% on development set, as observed in Table 1. The function received the drug entities to be evaluated e1 and e2, the list of every drug entity in the sentence and the CoreNLP dependency tree. This set of features was composed of:

- lib: lemma of the first token after e1 entity head
- wib: lemma of the first token after e2 entity head
- lpig: lemma and PoS tag of the first token after drug1 entity head
- eib: True/False. Presence of a third drug entity between e1 and e2.
- path: path with lemmas and relations from e1 head to LCS to e2 head
- path1: path up with lemmas and relations from e1 head to LCS
- path2: path down with lemmas and relations from LCS to e2 head

Where the head token for each gold entity refers to the token in the dependency parse tree that corresponds to the main noun or verb that represents the entity. See that for some reason, apart from the paths, we only add specific information about the next token after the first entity head. We also seem to miss many of the possibilities for features presented before, given the dependency object.

Individual token features

We designed a set of tests to assess the effects of adding the rest of a particular token information (word, lemma, PoS tag, and combinations). This meant that for every sentence, we created features such as "*wb_before_1=as*" or "*ltb_before_1=as_IN*", where the former saves the literal word "as" and the latter concatenates the lemma of "as" with its PoS tag. The "1" indicates here that "as" was the first word before the entity2. In the cases of "between" and "after", the numeration criteria starts referencing from entity 1 and entity 2, respectively, as seen in the Code.

This was done for every word in the sentence besides the entities, which created a huge feature space and delayed training set feature extraction considerably. Nonetheless, this would be worth it if it translated into a better classification of interactions. It did not in all our tests, decreasing macro F1 from 35% to between 31% to 34% for NB depending on the number of features we decided to add or not. In Table 2 we see the evaluation stats for the test with the original set of features + all individual features. It was clear to us that in this case the tradeoff between the complexity of the feature space and added information was not positive, for which we decided that this feature space should not be present in our best feature extractor.

As opposed to that, we found that the addition of the drug entities 1 and 2 POS tag in the form of "*e1_POStag=NN*" or "*e2_POStag=NN*" improved 1% or 2% the results on the majority of *devel* tests, and they included in our optimal extractor.

Paths extension

Original features contained paths with lemmas and dependency relations, so we decided to extend that with path variations that could potentially enrich the data space. We created features "*path1_nodes*", "*path2_nodes*", "*path1_edges*" "*path1_nodes*", "*path_edges*" and "*path_edges*" where, for the three paths we had in the default features it creates the versions based purely on words and PoS tags and for dependency relations. The addition of these features gave us a 3% increase on average in the macro F1 in the tests we performed. Two examples of these new features are:

- *path2_nodes=inhibitor<inhibitor<NN<metabolism< metabolism<NN<theophylline<theophylline<NN*
- *path_edges=conj<<<indir<conj<<<indir*

We believe that these path variations worked well to enhance classification because they add a less sparse and more specific approach compared to the other paths we had, which complements the feature set.

Clue lemmas

We analyzed the meaning behind the four different types of interactions to see if there was any pattern or possible indicators in the sentences that we could use in our favor to differentiate between them, rather than more analytical approaches as presented before. Here is the definition of every interaction type:

- **Advice**: DDI in which a recommendation or advice regarding the concomitant use of two drugs involved in them is described.
  - e.g.: "Interactions may be expected, and UROXATRAL should NOT be used in combination with other alpha-blockers."
- **Effect**: DDI in which the effect of the drug-drug interaction is described.
  - e.g.: "Aspirin and Paracetamol help decrease temperature."
- **Mechanism**: The mechanism of interaction can be pharmacodynamic (the effects of one drug are changed by the presence of another drug at its site of action, for example, "alcohol potentiates the depressor effect of barbiturates") pharmacokinetic (the processes by which drugs are absorbed, distributed, metabolized and excreted are affected, for example, ("induced the metabolism of", "increased the clearance of'). As already noted, a pharmacodynamic relationship between entities must be considered type effect.
  - e.g.: Grepafloxacin, like other quinolones, may inhibit the metabolism of caffeine and theobromine.
- **int**: the sentence simply states that an interaction occurs and does not provide any information about the interaction.
  - e.g.: The interaction of omeprazole and ketoconazole has been established.

We then believed that the presence of certain clue verbs before, between, and after the drug entities could be useful information for labeling the different types of interactions. Consequently, the first approach taken was to have a list of clue verbs for every interaction, which in the feature extractor would be triggered if a verb was in the list. Then we obtained features such as "*increase_clue_verb_between_entities=True"* and so. Even though it seemed a good idea beforehand, it had many issues. We realized that many verbs were equally important in different interaction classes, such as *increase* and *decrease*, which were frequent in both *mechanism* and *effect* interactions. In addition, some of the key tokens were not verbs. *Should* is a very informative word in the DrugBank dataset for *advice* interactions, but *should not* cancels that positive interaction regularly. Also, *interaction* and *interact* are often present in *int* sentences. In terms of results, our issues in separating key terms of the classes translated to the macro F1 score, with 0 to almost no effect on it. There we also found out that we would need to lemmatize the clue terms list, so there are more positive matches in the sentences.

This led to our second and final approach, the "*clue lemmas list"*. Now it was a unique list with no class division, with all the lemmas of the words that we believed were more important for interactions after thorough analysis. For every positive match in a word in a sentence, a feature would be created with a "True" per word + if before, between, or after the entities. For instance, the feature *"increase_after=True"* indicates that the lemma *increase* is present in the sentence after the two entities. Besides, "*should_between=False"* informs that the clue lemma *should* is not present between the entities evaluated in the sentence for interaction. Note that in the cases where we do not have any clue lemmas, the feature vector is very sparse. The addition of the clue lemmas features increased the macroaverage F1 from 35% with default features in Naive Bayes to 39.5% approximately, and joined with the path's feature extension to 42.9%. With the path's feature extension in the best hyperparametrization of Linear SVM it jumped to 48.8 % in the test dataset, as seen in Table 5.

Our list of key terms for the "*clue lemmas list"* is the following:

- clue_lemmas = ["affect", "effect", "diminish", "produce", "increase", "result", "decrease", "induce", "enhance", "lower", "cause", "interact", "interaction", "shall", "caution", "advise", "reduce", "prolong", "not"]

Final features

The final feature space contained:

- The default features
- Path extensions features
- Clue lemmas features
- POS tag of entities features

Experiments and Results

Here we summarize the most relevant results from many tests we undertook, for different feature combinations and parametrizations on the *devel* dataset, to the execution of evaluations on *test* dataset for the final results with the best hyperparametrization.

Naive Bayes

The best NB model with our best features was the one with alpha smoothing parameter = 0.1. In Tables 1 and 2, the results on the *devel* and *test* set for the beginning point are presented. Note the differences in the *int* class labeling, being in the *test* set the cause of a much lower macro average F1 score.

Table 1. Original feature set result on default NB, devel dataset:

	tp	fp	fn	#pred	#exp	P	R	F1
advise	69	293	72	362	141	19.1%	48.9%	27.4%
effect	102	144	210	246	312	41.5%	32.7%	36.6%
int	16	21	12	37	28	43.2%	57.1%	49.2%
mechanism	84	279	177	363	261	23.1%	32.2%	26.9%
M.avg	--	-	-	-		31.7%	42.7%	35.0%
m.avg	271	737	471	1008	742	26.9%	36.5%	31.0%
m.avg(no class)	435	573	307	1008	742	43.2%	58.6%	49.7%

Table 2. Original feature set result on default NB, test dataset:

	tp	fp	fn	#pred	#exp	P	R	F1
advise	83	244	126	327	209	25.4%	39.7%	31.0%
effect	110	226	176	336	286	32.7%	38.5%	35.4%
int	2	20	23	22	25	9.1%	8.0%	8.5%
mechanism	146	388	194	534	340	27.3%	42.9%	33.4%
M.avg	--	-	-	-		23.6%	32.3%	27.1%
m.avg	341	878	519	1219	860	28.0%	39.7%	32.8%
m.avg(no class)	487	732	373	1219	860	40.0%	56.6%	46.8%

In Table 3, as explained previously, we show that the effect of the huge individual token features worsens the results to 31%, with *int* as the class that causes the fraction of more new inaccurate predictions.

Table 3. Original + individual tokens feature set result on default NB, devel dataset:

	tp	fp	fn	#pred	#exp	P	R	F1
advise	107	308	34	415	141	25.8%	75.9%	38.5%
effect	148	240	164	388	312	38.1%	47.4%	42.3%
int	5	53	23	58	28	8.6%	17.9%	11.6%
mechanism	121	388	140	509	261	23.8%	46.4%	31.4%
M.avg	--	-	-	-		24.1%	46.9%	31.0%
m.avg	381	989	361	1370	742	27.8%	51.3%	36.1%
m.avg(no class)	476	894	266	1370	742	34.7%	64.2%	45.1%

Linear SVM

We applied grid parametrization to the final feature space and our best model was hyperparametrized with L2 regularization and a C regularization parameter of 0.2, although mean error values did not change much between different parametrizations.

Note that for the original feature set, Linear SVM results differ a lot from those ones from Table 2, for the Naive Bayes. We even start from a lower point in terms of overall results, with a 29.8% of macro average F1.

Table 4. Original feature set on best Linear SVC, test dataset:

	tp	fp	fn	#pred	#exp	P	R	F1
advise	10	4	199	14	209	71.4%	4.8%	9.0%
effect	49	17	237	66	286	74.2%	17.1%	27.8%
int	13	1	12	14	25	92.9%	52.0%	66.7%
mechanism	33	46	307	79	340	41.8%	9.7%	15.8%
M.avg	--	-	-	-		70.1%	20.9%	29.8%
m.avg	105	68	755	173	860	60.7%	12.2%	20.3%
m.avg(no class)	126	47	734	173	860	72.8%	14.7%	24.4%

One of the main outcomes is that our improvement in macroaverage F1 score is heavily weighted by our *int* class better classification results, with a 73.2% of F1 score on the *test* dataset. The reason behind that is the good effectiveness of the clue lemmas for the *int* sentences, with *interact* and *interaction* key terms very present in the DrugBank dataset fraction. We also managed to improve significantly the results for Linear SVM in the *advise*, *effect* and *mechanism* interactions as well, as a consequence of all features we added.

Table 5. Final feature set on best Linear SVC, test dataset:

	tp	fp	fn	#pred	#exp	P	R	F1
advise	47	31	162	78	209	60.3%	22.5%	32.8%
effect	103	71	183	174	286	59.2%	36.0%	44.8%
int	15	1	10	16	25	93.8%	60.0%	73.2%
mechanism	112	75	228	187	340	59.9%	32.9%	42.5%
M.avg	- -	-	-	-		68.3%	37.9%	48.3%
m.avg	277	178	583	455	860	60.9%	32.2%	42.1%
m.avg(no class)	337	118	523	455	860	74.1%	39.2%	51.3%

Conclusions

This project allowed us to learn about Relation Extraction NLP task, focused on the classification of drug-drug interactions. Through our experimentation, we discovered how dependency trees can be useful sources of information for feature extraction in sentences, and how using external resources like CoreNLP we can enrich our possibilities for building the feature space. This is a key step for the ML training for any algorithm, and we realized that it was more critical that any hyperparametrization we could apply to a particular model.

Furthermore, our experimentation allowed us to compare the performance of different machine learning algorithms for drug-drug interaction classification, Naive Bayes and Linear Support Vector Machines. We found that the linear SVM outperformed NB, achieving a macroaverage F1 score of 48.3% on the test set. This highlights the importance of selecting the appropriate machine learning algorithm for the task at hand, as well as the significance of feature extraction in NLP tasks like Relation Extraction. Overall, this project provided us with valuable insights into the complexities of drug-drug interaction classification and the importance of utilizing NLP techniques and machine learning algorithms to effectively address these challenges.

Code

Clue lemmas and *extract_features_final*

```python
clue_lemmas = ["affect", "effect", "diminish", "produce", "increase", "result",
"decrease", "induce", "enhance", "lower", "cause", "interact", "interaction", "shall",
"caution", "advise", "reduce", "prolong", "not"]
```

```
def extract_features_clue_lemmas_PAU(tree, entities, e1, e2, clue_lemmas):
    feats = set()
    total_tokens = tree.get_n_nodes()

    # get head token for each gold entity
    tkE1 = tree.get_fragment_head(entities[e1]['start'], entities[e1]['end'])
    tkE2 = tree.get_fragment_head(entities[e2]['start'], entities[e2]['end'])

    if tkE1 is not None and tkE2 is not None:  # else  should never happen since tree root
is always a common subsumer.)

        #POS TAG ENTITIES

        entity_num=1
        start1 = entities[e1]["start"]
        start2 = entities[e2]["start"]
        compro1 = False
        compro2 = False
        for n in tree.get_nodes():
            if start1 == tree.tree.nodes[n]["start"]:
                feats.add(f"e1_POStag=" + tree.tree.nodes[n]["tag"])
                compro1 = True
            elif start2 == tree.tree.nodes[n]["start"]:
                feats.add(f"e2_POStag=" + tree.tree.nodes[n]["tag"])
                compro2 = True
            elif compro1 and compro2:
                break

        # CHECK FOR PRESENCE BEFORE, BETWEEN AND AFTER OF CLUE LEMMAS
        for tk in range(0, tkE1):
            lemma = tree.get_lemma(tk).lower()
            if lemma in clue_lemmas:
                feats.add(lemma + f"_before=True")

        for tk in range(tkE1+1, tkE2):
            lemma = tree.get_lemma(tk).lower()
            if lemma in clue_lemmas:
                feats.add(lemma + f"_between=True")

        total_tokens = tree.get_n_nodes()
        for tk in range(tkE2+1, total_tokens):
            lemma = tree.get_lemma(tk).lower()
            if lemma in clue_lemmas:
                feats.add(lemma + f"_after=True")

        # for tk in range(tkE1+1, tkE2) :
        tk = tkE1 + 1
        try:
            while (tree.is_stopword(tk)):
                tk += 1
```

```
except:
    return set()
word = tree.get_word(tk)
lemma = tree.get_lemma(tk).lower()
tag = tree.get_tag(tk)
feats.add("lib=" + lemma)
feats.add("wib=" + word)
feats.add("lpib=" + lemma + "_" + tag)

eib = False
for tk in range(tkE1 + 1, tkE2):
    if tree.is_entity(tk, entities):
        eib = True

    # feature indicating the presence of an entity in between E1 and E2
feats.add('eib=' + str(eib))

# features about paths in the tree
lcs = tree.get_LCS(tkE1, tkE2)

path1 = tree.get_up_path(tkE1, lcs)
path1 = "<".join([tree.get_lemma(x) + "_" + tree.get_rel(x) for x in path1])
feats.add("path1=" + path1)

path2 = tree.get_down_path(lcs, tkE2)
path2 = ">".join([tree.get_lemma(x) + "_" + tree.get_rel(x) for x in path2])
feats.add("path2=" + path2)

path = path1 + "<" + tree.get_lemma(lcs) + "_" + tree.get_rel(lcs) + ">" + path2
feats.add("path=" + path)


#PATHS EXTENSION

path1 = tree.get_up_path(tkE1, lcs)
path1_nodes = []
path1_edges = []
for node in path1:
    path1_nodes.append(tree.get_word(node))
    path1_nodes.append(tree.get_lemma(node))
    path1_nodes.append(tree.get_tag(node))
    if node != lcs:
        parent = tree.get_parent(node)
        path1_edges.append(tree.get_rel(node))
        path1_edges.append('>' if parent == node + 1 else '<')
        path1_edges.append('dir' if parent == node + 1 else 'indir')
feats.add("path1_nodes=" + "<".join(path1_nodes))
feats.add("path1_edges=" + "<".join(path1_edges))

# path 2
```

```
        path2 = tree.get_down_path(lcs, tkE2)
        path2_nodes = []
        path2_edges = []
        for node in path2:
            path2_nodes.append(tree.get_word(node))
            path2_nodes.append(tree.get_lemma(node))
            path2_nodes.append(tree.get_tag(node))
            if node != lcs:
                parent = tree.get_parent(node)
                path2_edges.append(tree.get_rel(node))
                path2_edges.append('>' if parent == node + 1 else '<')
                path2_edges.append('dir' if parent == node + 1 else 'indir')
        feats.add("path2_nodes=" + "<".join(path2_nodes))
        feats.add("path2_edges=" + "<".join(path2_edges))

        # full path
        path = path1 + [lcs] + path2
        path_nodes = []
        path_edges = []
        for node in path:
            path_nodes.append(tree.get_word(node))
            path_nodes.append(tree.get_lemma(node))
            path_nodes.append(tree.get_tag(node))
            if node != lcs:
                parent = tree.get_parent(node)
                path_edges.append(tree.get_rel(node))
                path_edges.append('>' if parent == node + 1 else '<')
                path_edges.append('dir' if parent == node + 1 else 'indir')
        feats.add("path_nodes=" + "<".join(path_nodes))
        feats.add("path_edges=" + "<".join(path_edges))

        ##INDIVIDUAL FEATURES NOT ADDED IN OPTIMAL
        '''
        # loop before tkE1
        for c, tk in enumerate(range(0, tkE1)):
            # features for tokens before E1
            try:
                if tree.is_stopword(tk):
                    continue
                word = tree.get_word(tk).lower()
                lemma = tree.get_lemma(tk).lower()
                tag = tree.get_tag(tk)

                feats.add(f"wb_before_{tkE1 - c}=" + word)
                feats.add(f"lb_before_{tkE1 - c}=" + lemma)
                feats.add(f"tb_before_{tkE1 - c}=" + tag)
                feats.add(f"wlb_before_{tkE1 - c}=" + word + "_" + lemma)
                feats.add(f"wtb_before_{tkE1 - c}=" + word + "_" + tag)
                feats.add(f"ltb_before_{tkE1 - c}=" + lemma + "_" + tag)
                feats.add(f"wltb_before_{tkE1 - c}=" + word + "_" + lemma + "_" + tag)
                # Check for presence of clue verbs before tkE1
                for label, verb_list in zip(['moa', 'effect', 'advice', 'int'],
                                            [moa_clue_verbs, effect_clue_verbs,
advice_clue_verbs,
                                             int_clue_verbs]):
                    if word in verb_list:
                        feats.add(f'clue_verb_{label}_before={word}')
                        break
            except:
                return set()

        # loop between tkE1 an tkE2
        for c, tk in enumerate(range(tkE1 + 1, tkE2)):
            # features for tokens in between E1 and E2
            try:
                if tree.is_stopword(tk):
                    continue
                word = tree.get_word(tk).lower()

                # Remove any non-alphanumeric characters from the word
                # clean_word = ''.join(c for c in word if c.isalnum())

                lemma = tree.get_lemma(tk).lower()
                tag = tree.get_tag(tk)

                feats.add(f"wb_between_{c}=" + word)
                feats.add(f"lb_between_{c}=" + lemma)
                feats.add(f"tb_between_{c}=" + tag)
                feats.add(f"wlb_between_{c}=" + word + "_" + lemma)
                feats.add(f"wtb_between_{c}=" + word + "_" + tag)
                feats.add(f"ltb_between_{c}=" + lemma + "_" + tag)
                feats.add(f"wltb_between_{c}=" + word + "_" + lemma + "_" + tag)

                # Check for presence of clue verbs between entity 1 and entity 2
                for label, verb_list in zip(['moa', 'effect', 'advice', 'int'],
                                            [moa_clue_verbs, effect_clue_verbs,
advice_clue_verbs,
                                             int_clue_verbs]):
                    if word in verb_list:
                        feats.add(f'clue_verb_{label}_between={word}')
                        break
            except:
                return set()


        flag = False
        for c, tk in enumerate(range(tkE2 + 1, total_tokens)):
            # features for tokens after E2
            try:
                if tree.is_stopword(tk):
                    continue
                word = tree.get_word(tk).lower()
                lemma = tree.get_lemma(tk).lower()
                tag = tree.get_tag(tk)

                feats.add(f"wb_after_{c}=" + word)
                feats.add(f"lb_after_{c}=" + lemma)
                feats.add(f"tb_after_{c}=" + tag)
                feats.add(f"wlb_after_{c}=" + word + "_" + lemma)
                feats.add(f"wtb_after_{c}=" + word + "_" + tag)
                feats.add(f"ltb_after_{c}=" + lemma + "_" + tag)
                feats.add(f"wltb_after_{c}=" + word + "_" + lemma + "_" + tag)

                # Check for presence of clue verbs after entity 2
                for label, verb_list in zip(['moa', 'effect', 'advice', 'int'],
                                            [moa_clue_verbs, effect_clue_verbs,
advice_clue_verbs,
                                             int_clue_verbs]):
                    if word in verb_list:
                        feats.add(f'clue_verb_{label}_after={word}')
            except:
                return set()

        '''

    return feats
```