ELI5 Documentation

Release 0.1

Mikhail Korobov, Konstantin Lopuhin

November 25, 2016

Contents

1 Overview **3**
- 1.1 Installation . . . 3
- 1.2 Features . . . 3
- 1.3 Basic Usage . . . 3
- 1.4 Why? . . . 4
- 1.5 Architecture . . . 4

2 Tutorials **7**
- 2.1 Debugging scikit-learn text classification pipeline . . . 7
- 2.2 Named Entity Recognition using sklearn-crfsuite . . . 14

3 LIME **19**
- 3.1 Algorithm . . . 19
- 3.2 Caveats . . . 19
- 3.3 Alternative implementations . . . 20
- 3.4 Example: LIME on a synthetic data . . . 20
- 3.5 API . . . 20

4 API **21**
- 4.1 ELI5 top-level API . . . 21
- 4.2 eli5.formatters . . . 25
- 4.3 eli5.lightning . . . 26
- 4.4 eli5.lime . . . 26
- 4.5 eli5.sklearn . . . 28
- 4.6 eli5.sklearn_crfsuite . . . 31
- 4.7 eli5.base . . . 32

5 Contributing **37**

6 Changelog **39**
- 6.1 0.1.1 (2016-11-25) . . . 39
- 6.2 0.1 (2016-11-24) . . . 39
- 6.3 0.0.6 (2016-10-12) . . . 40
- 6.4 0.0.5 (2016-09-27) . . . 40
- 6.5 0.0.4 (2016-09-24) . . . 40
- 6.6 0.0.3 (2016-09-21) . . . 40
- 6.7 0.0.2 (2016-09-19) . . . 40
- 6.8 0.0.1 (2016-09-15) . . . 41

Python Module Index 43

ELI5 is a Python library which allows to visualize and debug various Machine Learning models using unified API. It has built-in support for several ML frameworks and provides a way to explain black-box models.

CHAPTER 1

Overview

1.1 Installation

ELI5 works in Python 2.7 and Python 3.4+. Currently it requires scikit-learn 0.18+, so make sure scikit-learn is installed first, then install eli5 using pip:

```
pip install 'scikit-learn > 0.18'
pip install eli5
```

1.2 Features

ELI5 is a Python package which helps to debug machine learning classifiers and explain their predictions. It provides support for the following machine learning frameworks and packages:

- scikit-learn. Currently ELI5 allows to explain weights and predictions of scikit-learn linear classifiers and regressors, print decision trees as text or as SVG, show feature importances of random forests. ELI5 understands text processing utilities from scikit-learn and can highlight text data accordingly. It also allows to debug scikit-learn pipelines which contain HashingVectorizer, by undoing hashing.
- lightning - explain weights and predictions of lightning classifiers and regressors.
- sklearn-crfsuite. ELI5 allows to check weights of sklearn_crfsuite.CRF models.

ELI5 also provides an alternative implementation of *LIME* algorithm, which allows to explain predictions of any black-box classifier. This feature is currently experimental.

Explanation and formatting are separated; you can get text-based explanation to display in console, HTML version embeddable in an IPython notebook or web dashboards, or JSON version which allows to implement custom rendering and formatting on a client.

1.3 Basic Usage

There are two main ways to look at a classification or a regression model:

1. inspect model parameters and try to figure out how the model works globally;
2. inspect an individual prediction of a model, try to figure out why the model makes the decision it makes.

For (1) ELI5 provides `eli5.show_weights()` function; for (2) it provides `eli5.show_prediction()` function.

If the ML library you're working with is supported then you usually can enter the following in IPython Notebook:

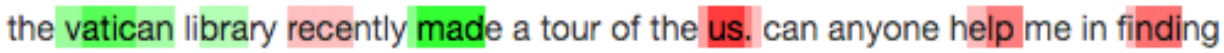
```
import eli5
eli5.explain_weights(clf)
```

and get an explanation like this:

Weight	Feature
+11.493	<BIAS>
+6.280	x
-14.140	y

Supported arguments and the exact way the classifier is visualized depends on a library.

To explain an individual prediction (2) use *eli5.show_prediction()* function. Exact parameters depend on a classifier and on input data kind (text, tabular, images). For example, you may get text highlighted like this if you're using one of the scikit-learn vectorizers with char ngrams:



To learn more, follow the *Tutorials*, check example IPython notebooks and read documentation sections specific to your framework.

1.4 Why?

For some of classifiers inspection and debugging is easy, for others this is hard. It is not a rocket science to take coefficients of a linear classifier, relate them to feature names and show in an HTML table. ELI5 aims to handle not only simple cases, but even for simple cases having a unified API for inspection has a value:

- you can call a ready-made function from ELI5 and get a nicely formatted result immediately;
- formatting code can be reused between machine learning frameworks;
- 'drill down' code like feature filtering or text highlighting can be reused;
- there are lots of gotchas and small differences which ELI5 takes care of;
- algorithms like *LIME* (paper) try to explain a black-box classifier through a locally-fit simple, interpretable classifier. It means that with each additional supported "simple" classifier/regressor algorithms like LIME are getting more options automatically.

1.5 Architecture

In ELI5 "explanation" is separated from output format: *eli5.explain_weights()* and *eli5.explain_prediction()* return *Explanation* instances; then functions from *eli5.formatters* can be used to get HTML, text or dict/JSON representation of the explanation.

It is not convenient to do that all when working interactively in IPython notebooks, so there are *eli5.show_weights()* and *eli5.show_prediction()* functions which do explanation and formatting in a single step.

Explain functions are not doing any work by themselves; they call a concrete implementation based on estimator type. So e.g. *eli5.explain_weights()* calls *eli5.sklearn.explain_weights.explain_linear_classifier_weights()* if `sklearn.linear_model.LogisticRegression` classifier is passed as an estimator.

CHAPTER 2

Tutorials

Note: This tutorial is intended to be run in an IPython notebook. It is also available as a notebook file here.

2.1 Debugging scikit-learn text classification pipeline

scikit-learn docs provide a nice text classification tutorial. Make sure to read it first. We'll be doing something similar to it, while taking more detailed look at classifier weights and predictions.

2.1.1 1. Baseline model

First, we need some data. Let's load 20 Newsgroups data, keeping only 4 categories:

```
from sklearn.datasets import fetch_20newsgroups

categories = ['alt.atheism', 'soc.religion.christian',
              'comp.graphics', 'sci.med']
twenty_train = fetch_20newsgroups(
    subset='train',
    categories=categories,
    shuffle=True,
    random_state=42
)
twenty_test = fetch_20newsgroups(
    subset='test',
    categories=categories,
    shuffle=True,
    random_state=42
)
```

A basic text processing pipeline - bag of words features and Logistic Regression as a classifier:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegressionCV
from sklearn.pipeline import make_pipeline

vec = CountVectorizer()
clf = LogisticRegressionCV()
pipe = make_pipeline(vec, clf)
pipe.fit(twenty_train.data, twenty_train.target);
```

We're using LogisticRegressionCV here to adjust regularization parameter C automatically. It allows to compare different vectorizers - optimal C value could be different for different input features (e.g. for bigrams or for character-level input). An alternative would be to use GridSearchCV or RandomizedSearchCV.

Let's check quality of this pipeline:

```
from sklearn import metrics

def print_report(pipe):
    y_test = twenty_test.target
    y_pred = pipe.predict(twenty_test.data)
    report = metrics.classification_report(y_test, y_pred,
        target_names=twenty_test.target_names)
    print(report)
    print("accuracy: {:0.3f}".format(metrics.accuracy_score(y_test, y_pred)))

print_report(pipe)
```

```
                          precision    recall  f1-score   support

           alt.atheism       0.93      0.80      0.86       319
         comp.graphics       0.87      0.96      0.91       389
               sci.med       0.94      0.81      0.87       396
soc.religion.christian       0.85      0.98      0.91       398

           avg / total       0.90      0.89      0.89      1502

accuracy: 0.891
```

Not bad. We can try other classifiers and preprocessing methods, but let's check first what the model learned using *eli5.show_weights()* function:

```
import eli5
eli5.show_weights(clf, top=10)
```

The table above doesn't make any sense; the problem is that eli5 was not able to get feature and class names from the classifier object alone. We can provide feature and target names explicitly:

```
# eli5.show_weights(clf,
#                   feature_names=vec.get_feature_names(),
#                   target_names=twenty_test.target_names)
```

The code above works, but a better way is to provide vectorizer instead and let eli5 figure out the details automatically:

```
eli5.show_weights(clf, vec=vec, top=10,
                  target_names=twenty_test.target_names)
```

This starts to make more sense. Columns are target classes. In each column there are features and their weights. Intercept (bias) feature is shown as <BIAS> in the same table. We can inspect features and weights because we're using a bag-of-words vectorizer and a linear classifier (so there is a direct mapping between individual words and classifier coefficients). For other classifiers features can be harder to inspect.

Some features look good, but some don't. It seems model learned some names specific to a dataset (email parts, etc.) though, instead of learning topic-specific words. Let's check prediction results on an example:

```
eli5.show_prediction(clf, twenty_test.data[0], vec=vec,
                     target_names=twenty_test.target_names)
```

What can be highlighted in text is highlighted in text. There is also a separate table for features which can't be highlighted in text - <BIAS> in this case. If you hover mouse on a highlighted word it shows you a weight of this word in a title. Words are colored according to their weights.

2.1.2 2. Baseline model, improved data

Aha, from the highlighting above it can be seen that a classifier learned some non-interesting stuff indeed, e.g. it remembered parts of email addresses. We should probably clean the data first to make it more interesting; improving model (trying different classifiers, etc.) doesn't make sense at this point - it may just learn to leverage these email addresses better.

In practice we'd have to do cleaning yourselves; in this example 20 newsgroups dataset provides an option to remove footers and headers from the messages. Nice. Let's clean up the data and re-train a classifier.

```
twenty_train = fetch_20newsgroups(
    subset='train',
    categories=categories,
    shuffle=True,
    random_state=42,
    remove=['headers', 'footers'],
)
twenty_test = fetch_20newsgroups(
    subset='test',
    categories=categories,
    shuffle=True,
    random_state=42,
    remove=['headers', 'footers'],
)

vec = CountVectorizer()
clf = LogisticRegressionCV()
pipe = make_pipeline(vec, clf)
pipe.fit(twenty_train.data, twenty_train.target);
```

We just made the task harder and more realistic for a classifier.

```
print_report(pipe)
```

```
                         precision    recall  f1-score   support

           alt.atheism       0.83      0.78      0.80       319
         comp.graphics       0.82      0.96      0.88       389
               sci.med       0.89      0.80      0.84       396
soc.religion.christian       0.88      0.86      0.87       398

           avg / total       0.85      0.85      0.85      1502

accuracy: 0.852
```

A great result - we just made quality worse! Does it mean pipeline is worse now? No, likely it has a better quality on unseen messages. It is evaluation which is more fair now. Inspecting features used by classifier allowed us to notice a problem with the data and made a good change, despite of numbers which told us not to do that.

Instead of removing headers and footers we could have improved evaluation setup directly, using e.g. GroupKFold from scikit-learn. Then quality of old model would have dropped, we could have removed headers/footers and see increased accuracy, so the numbers would have told us to remove headers and footers. It is not obvious how to split data though, what groups to use with GroupKFold.

So, what have the updated classifier learned? (output is less verbose because only a subset of classes is shown - see "targets" argument):

```
eli5.show_prediction(clf, twenty_test.data[0], vec=vec,
                     target_names=twenty_test.target_names,
                     targets=['sci.med'])
```

Hm, it no longer uses email addresses, but it still doesn't look good: classifier assigns high weights to seemingly unrelated words like 'do' or 'my'. These words appear in many texts, so maybe classifier uses them as a proxy for bias. Or maybe some of them are more common in some of classes.

2.1.3 3. Pipeline improvements

To help classifier we may filter out stop words:

```
vec = CountVectorizer(stop_words='english')
clf = LogisticRegressionCV()
pipe = make_pipeline(vec, clf)
pipe.fit(twenty_train.data, twenty_train.target)

print_report(pipe)
```

```
                         precision    recall  f1-score   support

           alt.atheism       0.87      0.76      0.81       319
         comp.graphics       0.85      0.95      0.90       389
               sci.med       0.93      0.85      0.89       396
soc.religion.christian       0.85      0.89      0.87       398

           avg / total       0.87      0.87      0.87      1502

accuracy: 0.871
```

```
eli5.show_prediction(clf, twenty_test.data[0], vec=vec,
                     target_names=twenty_test.target_names,
                     targets=['sci.med'])
```

Looks better, isn't it?

Alternatively, we can use TF*IDF scheme; it should give a somewhat similar effect.

Note that we're cross-validating LogisticRegression regularisation parameter here, like in other examples (LogisticRegressionCV, not LogisticRegression). TF*IDF values are different from word count values, so optimal C value can be different. We could draw a wrong conclusion if a classifier with fixed regularization strength is used - the chosen C value could have worked better for one kind of data.

```
from sklearn.feature_extraction.text import TfidfVectorizer

vec = TfidfVectorizer()
clf = LogisticRegressionCV()
pipe = make_pipeline(vec, clf)
pipe.fit(twenty_train.data, twenty_train.target)

print_report(pipe)
```

```
                         precision    recall  f1-score   support

           alt.atheism       0.91      0.79      0.85       319
```

```
         comp.graphics       0.83      0.97      0.90       389
               sci.med       0.95      0.87      0.91       396
soc.religion.christian       0.90      0.91      0.91       398

           avg / total       0.90      0.89      0.89      1502

accuracy: 0.892
```

```
eli5.show_prediction(clf, twenty_test.data[0], vec=vec,
                     target_names=twenty_test.target_names,
                     targets=['sci.med'])
```

It helped, but didn't have quite the same effect. Why not do both?

```
vec = TfidfVectorizer(stop_words='english')
clf = LogisticRegressionCV()
pipe = make_pipeline(vec, clf)
pipe.fit(twenty_train.data, twenty_train.target)

print_report(pipe)
```

```
                         precision    recall  f1-score   support

           alt.atheism       0.93      0.77      0.84       319
         comp.graphics       0.84      0.97      0.90       389
               sci.med       0.95      0.89      0.92       396
soc.religion.christian       0.88      0.92      0.90       398

           avg / total       0.90      0.89      0.89      1502

accuracy: 0.893
```

```
eli5.show_prediction(clf, twenty_test.data[0], vec=vec,
                     target_names=twenty_test.target_names,
                     targets=['sci.med'])
```

This starts to look good!

2.1.4 4. Char-based pipeline

Maybe we can get somewhat better quality by choosing a different classifier, but let's skip it for now. Let's try other analysers instead - use char n-grams instead of words:

```
vec = TfidfVectorizer(stop_words='english', analyzer='char',
                      ngram_range=(3,5))
clf = LogisticRegressionCV()
pipe = make_pipeline(vec, clf)
pipe.fit(twenty_train.data, twenty_train.target)

print_report(pipe)
```

```
                         precision    recall  f1-score   support

           alt.atheism       0.93      0.79      0.85       319
         comp.graphics       0.81      0.97      0.89       389
               sci.med       0.95      0.86      0.90       396
soc.religion.christian       0.89      0.91      0.90       398
```

```
           avg / total       0.89      0.89      0.89      1502

accuracy: 0.888
```

```
eli5.show_prediction(clf, twenty_test.data[0], vec=vec,
                     target_names=twenty_test.target_names)
```

It works, but quality is a bit worse. Also, it takes ages to train.

It looks like stop_words have no effect now - in fact, this is documented in scikit-learn docs, so our stop_words='english' was useless. But at least it is now more obvious how the text looks like for a char ngram-based classifier. Grab a cup of tea and see how char_wb looks like:

```
vec = TfidfVectorizer(analyzer='char_wb', ngram_range=(3,5))
clf = LogisticRegressionCV()
pipe = make_pipeline(vec, clf)
pipe.fit(twenty_train.data, twenty_train.target)

print_report(pipe)
```

```
                         precision    recall  f1-score   support

           alt.atheism       0.93      0.79      0.85       319
         comp.graphics       0.87      0.96      0.91       389
               sci.med       0.91      0.90      0.90       396
soc.religion.christian       0.89      0.91      0.90       398

           avg / total       0.90      0.89      0.89      1502

accuracy: 0.894
```

```
eli5.show_prediction(clf, twenty_test.data[0], vec=vec,
                     target_names=twenty_test.target_names)
```

The result is similar, with some minor changes. Quality is better for unknown reason; maybe cross-word dependencies are not that important.

2.1.5 5. Debugging HashingVectorizer

To check that we can try fitting word n-grams instead of char n-grams. But let's deal with efficiency first. To handle large vocabularies we can use HashingVectorizer from scikit-learn; to make training faster we can employ SGDCLassifier:

```
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.linear_model import SGDClassifier

vec = HashingVectorizer(stop_words='english', ngram_range=(1,2))
clf = SGDClassifier(n_iter=10, random_state=42)
pipe = make_pipeline(vec, clf)
pipe.fit(twenty_train.data, twenty_train.target)

print_report(pipe)
```

```
                         precision    recall  f1-score   support

           alt.atheism       0.90      0.80      0.85       319
         comp.graphics       0.88      0.96      0.92       389
```

```
               sci.med       0.93      0.90      0.92       396
soc.religion.christian       0.89      0.91      0.90       398

           avg / total       0.90      0.90      0.90      1502

accuracy: 0.899
```

It was super-fast! We're not choosing regularization parameter using cross-validation though. Let's check what model learned:

```
eli5.show_prediction(clf, twenty_test.data[0], vec=vec,
                     target_names=twenty_test.target_names,
                     targets=['sci.med'])
```

Result looks similar to CountVectorizer. But with HashingVectorizer we don't even have a vocabulary! Why does it work?

```
eli5.show_weights(clf, vec=vec, top=10,
                  target_names=twenty_test.target_names)
```

Ok, we don't have a vocabulary, so we don't have feature names. Are we out of luck? Nope, eli5 has an answer for that: *InvertableHashingVectorizer* It can be used to get feature names for HahshingVectorizer without fitiing a huge vocabulary. It still needs some data to learn words -> hashes mapping though; we can use a random subset of data to fit it.

```
from eli5.sklearn import InvertableHashingVectorizer
import numpy as np
```

```
ivec = InvertableHashingVectorizer(vec)
sample_size = len(twenty_train.data) // 10
X_sample = np.random.choice(twenty_train.data, size=sample_size)
ivec.fit(X_sample);
```

```
eli5.show_weights(clf, vec=ivec, top=20,
                  target_names=twenty_test.target_names)
```

There are collisions (hover mouse over features with "..."), and there are important features which were not seen in the random sample (FEATURE[...]), but overall it looks fine.

"rutgers edu" bigram feature is suspicious though, it looks like a part of URL.

```
rutgers_example = [x for x in twenty_train.data if 'rutgers' in x.lower()][0]
print(rutgers_example)
```

```
In article <Apr.8.00.57.41.1993.28246@athos.rutgers.edu> REXLEX@fnal.gov writes:
>In article <Apr.7.01.56.56.1993.22824@athos.rutgers.edu> shrum@hpfcso.fc.hp.com
>Matt. 22:9-14 'Go therefore to the main highways, and as many as you find
>there, invite to the wedding feast.'...

>hmmmmmm.  Sounds like your theology and Christ's are at odds. Which one am I
>to believe?
```

Yep, it looks like model learned this address instead of learning something useful.

```
eli5.show_prediction(clf, rutgers_example, vec=vec,
                     target_names=twenty_test.target_names,
                     targets=['soc.religion.christian'])
```

Quoted text makes it too easy for model to classify some of the messages; that won't generalize to new messages. So to improve the model next step could be to process the data further, e.g. remove quoted text or replace email addresses

with a special token.

You get the idea: looking at features helps to understand how classifier works. Maybe even more importantly, it helps to notice preprocessing bugs, data leaks, issues with task specification - all these nasty problems you get in a real world.

Note: This tutorial is intended to be run in an IPython notebook. It is also available as a notebook file here.

2.2 Named Entity Recognition using sklearn-crfsuite

In this notebook we train a basic CRF model for Named Entity Recognition on CoNLL2002 data (following https://github.com/TeamHG-Memex/sklearn-crfsuite/blob/master/docs/CoNLL2002.ipynb) and check its weights to see what it learned.

To follow this tutorial you need NLTK > 3.x and sklearn-crfsuite Python packages. The tutorial uses Python 3.

```
import nltk
import sklearn_crfsuite
import eli5
```

2.2.1 1. Training data

CoNLL 2002 datasets contains a list of Spanish sentences, with Named Entities annotated. It uses IOB2 encoding. CoNLL 2002 data also provide POS tags.

```
train_sents = list(nltk.corpus.conll2002.iob_sents('esp.train'))
test_sents = list(nltk.corpus.conll2002.iob_sents('esp.testb'))
train_sents[0]
```

```
[('Melbourne', 'NP', 'B-LOC'),
 ('(', 'Fpa', 'O'),
 ('Australia', 'NP', 'B-LOC'),
 (')', 'Fpt', 'O'),
 (',', 'Fc', 'O'),
 ('25', 'Z', 'O'),
 ('may', 'NC', 'O'),
 ('(', 'Fpa', 'O'),
 ('EFE', 'NC', 'B-ORG'),
 (')', 'Fpt', 'O'),
 ('.', 'Fp', 'O')]
```

2.2.2 2. Feature extraction

POS tags can be seen as pre-extracted features. Let's extract more features (word parts, simplified POS tags, lower/title/upper flags, features of nearby words) and convert them to sklear-crfsuite format - each sentence should be converted to a list of dicts. This is a very simple baseline; you certainly can do better.

```
def word2features(sent, i):
    word = sent[i][0]
    postag = sent[i][1]

    features = {
```

```
        'bias': 1.0,
        'word.lower()': word.lower(),
        'word[-3:]': word[-3:],
        'word.isupper()': word.isupper(),
        'word.istitle()': word.istitle(),
        'word.isdigit()': word.isdigit(),
        'postag': postag,
        'postag[:2]': postag[:2],
    }
    if i > 0:
        word1 = sent[i-1][0]
        postag1 = sent[i-1][1]
        features.update({
            '-1:word.lower()': word1.lower(),
            '-1:word.istitle()': word1.istitle(),
            '-1:word.isupper()': word1.isupper(),
            '-1:postag': postag1,
            '-1:postag[:2]': postag1[:2],
        })
    else:
        features['BOS'] = True

    if i < len(sent)-1:
        word1 = sent[i+1][0]
        postag1 = sent[i+1][1]
        features.update({
            '+1:word.lower()': word1.lower(),
            '+1:word.istitle()': word1.istitle(),
            '+1:word.isupper()': word1.isupper(),
            '+1:postag': postag1,
            '+1:postag[:2]': postag1[:2],
        })
    else:
        features['EOS'] = True

    return features


def sent2features(sent):
    return [word2features(sent, i) for i in range(len(sent))]

def sent2labels(sent):
    return [label for token, postag, label in sent]

def sent2tokens(sent):
    return [token for token, postag, label in sent]

X_train = [sent2features(s) for s in train_sents]
y_train = [sent2labels(s) for s in train_sents]

X_test = [sent2features(s) for s in test_sents]
y_test = [sent2labels(s) for s in test_sents]
```

This is how features extracted from a single token look like:

```
X_train[0][1]
```

```
{'+1:postag': 'NP',
 '+1:postag[:2]': 'NP',
 '+1:word.istitle()': True,
 '+1:word.isupper()': False,
 '+1:word.lower()': 'australia',
 '-1:postag': 'NP',
 '-1:postag[:2]': 'NP',
 '-1:word.istitle()': True,
 '-1:word.isupper()': False,
 '-1:word.lower()': 'melbourne',
 'bias': 1.0,
 'postag': 'Fpa',
 'postag[:2]': 'Fp',
 'word.isdigit()': False,
 'word.istitle()': False,
 'word.isupper()': False,
 'word.lower()': '(',
 'word[-3:]': '('}
```

2.2.3 3. Train a CRF model

Once we have features in a right format we can train a linear-chain CRF (Conditional Random Fields) model using sklearn_crfsuite.CRF:

```
crf = sklearn_crfsuite.CRF(
    algorithm='lbfgs',
    c1=0.1,
    c2=0.1,
    max_iterations=20,
    all_possible_transitions=False,
)
crf.fit(X_train, y_train);
```

2.2.4 4. Inspect model weights

CRFsuite CRF models use two kinds of features: state features and transition features. Let's check their weights using eli5.explain_weights:

```
eli5.show_weights(crf, top=30)
```

Features don't use gazetteers, so model had to remember some geographic names from the training data, e.g. that España is a location.

Transition features make sense: at least model learned that I-ENITITY must follow B-ENTITY, and that some transitions are unlikely, e.g. it is not common to have location right after an organization name (I-LOC -> B-ORG has a large negative weight).

We'd also expect that O -> I-ENTIRY transitions have large negative weights because they are impossible, but these transitions have zero weight, not negative weight; it can be a problem, and decrease quality. sklearn_crfsuite.CRF provides `all_possible_transitions` argument which allows model to learn weights for transitions which are not observed in training data. Let's check how does it affect the result:

```
crf = sklearn_crfsuite.CRF(
    algorithm='lbfgs',
    c1=0.1,
    c2=0.1,
```

```
    max_iterations=20,
    all_possible_transitions=True,
)
crf.fit(X_train, y_train);
```

```
eli5.show_weights(crf, top=5, show=['transition_features'])
```

With `all_possible_transitions=True` CRF learned large negative weights for impossible transitions like O -> I-ORG.

2.2.5 5. Customization

The table above is large and kind of hard to inspect; eli5 provides several options to look only at a part of features. You can check only a subset of labels:

```
eli5.show_weights(crf, top=10, targets=['O', 'B-ORG', 'I-ORG'])
```

Another option is to check only some of the features - it helps to check if a feature function works as intended. For example, let's check how word shape features are used by model using `feature_re` argument and hide transition table:

```
eli5.show_weights(crf, top=10, feature_re='^word\.is',
                  horizontal_layout=False, show=['targets'])
```

Looks fine - UPPERCASE and Titlecase words are likely to be entities of some kind.

2.2.6 6. Formatting in console

It is also possible to format the result as text (could be useful in console):

```
expl = eli5.explain_weights(crf, top=5, targets=['O', 'B-LOC', 'I-LOC'])
print(eli5.format_as_text(expl))
```

```
Explained as: CRF

Transition features:
            O    B-LOC    I-LOC
-----  ------  -------  -------
O       2.732    1.217   -4.675
B-LOC  -0.226   -0.091    3.378
I-LOC  -0.184   -0.585    2.404

y='O' top features
---------------------------
  +4.931  BOS
  +3.754  postag[:2]:Fp
  +3.539  bias
       ...  (15043 more positive features)
       ...  (3906 more negative features)
  -3.685  word.isupper()
  -7.025  word.istitle()

y='B-LOC' top features
---------------------------
  +2.397  word.istitle()
  +2.147  -1:word.lower():en
```

```
       ...  (2284 more positive features)
       ...  (433 more negative features)
  -1.080  postag:SP
  -1.080  postag[:2]:SP
  -1.273  -1:word.istitle()

y='I-LOC' top features
---------------------------
  +0.882  -1:word.lower():de
  +0.780  -1:word.istitle()
  +0.718  word[-3:]:de
  +0.711  word.lower():de
       ...  (1684 more positive features)
       ...  (268 more negative features)
  -1.965  BOS
```

CHAPTER 3

LIME

Warning: eli5 LIME implementation is experimental and unchecked on real tasks. It will be improved in future.

3.1 Algorithm

LIME (Ribeiro et. al. 2016) is an algorithm to explain predictions of black-box estimators:

1. Generate a fake dataset from the example we're going to explain.
2. Use black-box estimator to get target values for each example in a generated dataset (e.g. class probabilities).
3. Train a new white-box estimator, using generated dataset and generated labels as training data. It means we're trying to create an estimator which works the same as a black-box estimator, but which is easier to inspect. It doesn't have to work well globally, but it must approximate the black-box model well in the area close to the original example.

   To express "area close to the original example" user must provide a distance/similarity metric for examples in a generated dataset. Then training data is weighted according to a distance from the original example - the further is example, the less it affects weights of a white-box estimator.
4. Explain the original example through weights of this white-box estimator instead.
5. Prediction quality of a white-box classifer shows how well it approximates the black-box classifier. If the quality is low then explanation shouldn't be trusted.

3.2 Caveats

It sounds too good to be true, and indeed there are caveats:

1. If a white-box estimator gets a high score on a generated dataset it doesn't necessarily mean it could be trusted - it could also mean that the generated dataset is too easy and uniform, or that similarity metric provided by user assigns very low values for most examples, so that "area close to the original example" is too small to be interesting.
2. Fake dataset generation is the main issue; it is task-specific to a large extent. So LIME can work with any black-box classifier, but user may need to write code specific for each dataset. There is an opposite tradeoff in inspecting model weights: it works for any task, but one must write inspection code for each estimator type.

   eli5.lime provides dataset generation utilities for text data (remove random words) and for arbitrary data (sampling using Kernel Density Estimation).

3. Similarity metric has a huge effect on a result. By choosing neighbourhood of a different size one can get opposite explanations.

3.3 Alternative implementations

There is a LIME implementation by LIME authors: https://github.com/marcotcr/lime, so it is eli5.lime which should be considered as alternative. At the time of writing eli5.lime has some differences from the canonical LIME implementation:

1. eli5 supports many white-box classifiers from several libraries, you can use any of them with LIME;
2. eli5 supports dataset generation using Kernel Density Estimation, to ensure that generated dataset looks similar to the original dataset;
3. for explaining predictions of probabilistic classifiers eli5 uses another classifier by default, trained using cross-entropy loss, while canonical library fits regression model on probability output.

There are also features which are supported by original implementation, but not by eli5, and the UIs are different. Currently ELI5 implementation is experimental and not battle-tested.

3.4 Example: LIME on a synthetic data

See this IPython notebook.

3.5 API

See *eli5.lime*.

CHAPTER 4

API

API documentation is auto-generated.

4.1 ELI5 top-level API

The following functions are exposed to a top level, e.g. `eli5.explain_weights`.

explain_weights(**args*, ***kw*)

Return an explanation of estimator parameters (weights).

explain_weights() is not doing any work itself, it dispatches to a concrete implementation based on estimator type.

Parameters

- **estimator** (*object*) – Estimator instance. This argument must be positional.
- **top** (*int or (int, int) tuple, optional*) – Number of features to show. When `top` is int, `top` features with a highest absolute values are shown. When it is (pos, neg) tuple, no more than `pos` positive features and no more than `neg` negative features is shown. `None` value means no limit.

  This argument may be supported or not, depending on estimator type.
- **target_names** (*list[str] or {'old_name': 'new_name'} dict, optional*) – Names of targets or classes. This argument can be used to provide human-readable class/target names for estimators which don't expose clss names themselves. It can be also used to rename estimator-provided classes before displaying them.

  This argument may be supported or not, depending on estimator type.
- **targets** (*list, optional*) – Order of class/target names to show. This argument can be also used to show information only for a subset of classes. It should be a list of class / target names which match either names provided by an estimator or names defined in `target_names` parameter.

  This argument may be supported or not, depending on estimator type.
- **feature_names** (*list, optional*) – A list of feature names. It allows to specify feature names when they are not provided by an estimator object.

  This argument may be supported or not, depending on estimator type.
- **feature_re** (*str, optional*) – Only feature names which match `feature_re` regex are returned.

- ****kwargs** (*dict*) – Keyword arguments. All keyword arguments are passed to concrete explain_weights... implementations.

Returns

Explanation – `Explanation` result. Use one of the formatting functions from *eli5.formatters* to print it in a human-readable form.

Explanation instances have repr which works well with IPython notebook, but it can be a better idea to use *eli5.show_weights()* instead of *eli5.explain_weights()* if you work with IPython: *eli5.show_weights()* allows to customize formatting without a need to import *eli5.formatters* functions.

explain_prediction(**args*, ***kw*)

Return an explanation of an estimator prediction.

explain_prediction() is not doing any work itself, it dispatches to a concrete implementation based on estimator type.

Parameters

- **estimator** (*object*) – Estimator instance. This argument must be positional.
- **doc** (*object*) – Example to run estimator on. Estimator makes a prediction for this example, and *explain_prediction()* tries to show information about this prediction.
- **top** (*int or (int, int) tuple, optional*) – Number of features to show. When `top` is int, `top` features with a highest absolute values are shown. When it is (pos, neg) tuple, no more than `pos` positive features and no more than `neg` negative features is shown. `None` value means no limit (default).

  This argument may be supported or not, depending on estimator type.
- **target_names** (*list[str] or {'old_name': 'new_name'} dict, optional*) – Names of targets or classes. This argument can be used to provide human-readable class/target names for estimators which don't expose clss names themselves. It can be also used to rename estimator-provided classes before displaying them.

  This argument may be supported or not, depending on estimator type.
- **targets** (*list, optional*) – Order of class/target names to show. This argument can be also used to show information only for a subset of classes. It should be a list of class / target names which match either names provided by an estimator or names defined in `target_names` parameter.

  This argument may be supported or not, depending on estimator type.
- **feature_names** (*list, optional*) – A list of feature names. It allows to specify feature names when they are not provided by an estimator object.

  This argument may be supported or not, depending on estimator type.
- ****kwargs** (*dict*) – Keyword arguments. All keyword arguments are passed to concrete explain_prediction... implementations.

Returns

Explanation – *Explanation* result. Use one of the formatting functions from *eli5.formatters* to print it in a human-readable form.

Explanation instances have repr which works well with IPython notebook, but it can be a better idea to use *eli5.show_prediction()* instead of *eli5.explain_prediction()* if you work with IPython: *eli5.show_prediction()* allows to customize formatting without a need to import *eli5.formatters* functions.

show_weights(*estimator*, ***kwargs*)

Return an explanation of estimator parameters (weights) as an IPython.display.HTML object. Use this function to show classifier weights in IPython.

`show_weights()` accepts all `eli5.explain_weights()` arguments and all `eli5.formatters.html.format_as_html()` keyword arguments, so it is possible to get explanation and customize formatting in a single call.

Parameters

- **estimator** (*object*) – Estimator instance. This argument must be positional.
- **top** (*int or (int, int) tuple, optional*) – Number of features to show. When `top` is int, `top` features with a highest absolute values are shown. When it is (pos, neg) tuple, no more than `pos` positive features and no more than `neg` negative features is shown. `None` value means no limit.

  This argument may be supported or not, depending on estimator type.
- **target_names** (*list[str] or {'old_name': 'new_name'} dict, optional*) – Names of targets or classes. This argument can be used to provide human-readable class/target names for estimators which don't expose clss names themselves. It can be also used to rename estimator-provided classes before displaying them.

  This argument may be supported or not, depending on estimator type.
- **targets** (*list, optional*) – Order of class/target names to show. This argument can be also used to show information only for a subset of classes. It should be a list of class / target names which match either names provided by an estimator or names defined in `target_names` parameter.

  This argument may be supported or not, depending on estimator type.
- **feature_names** (*list, optional*) – A list of feature names. It allows to specify feature names when they are not provided by an estimator object.

  This argument may be supported or not, depending on estimator type.
- **feature_re** (*str, optional*) – Only feature names which match `feature_re` regex are returned.
- **show** (*List[str], optional*) – List of sections to show. Allowed values:
  - 'targets' - per-target feature weights;
  - 'transition_features' - transition features of a CRF model;
  - 'feature_importances' - feature importances of a decision tree or an ensemble-based estimator;
  - 'decision_tree' - decision tree in a graphical form;
  - 'method' - a string with explanation method;
  - 'description' - description of explanation method and its caveats.
- **horizontal_layout** (*bool*) – When True, feature weight tables are printed horizontally (left to right); when False, feature weight tables are printed vertically (top to down). Default is True.
- **highlight_spaces** (*bool or None, optional*) – Whether to highlight spaces in feature names. This is useful if you work with text and have ngram features which may include spaces at left or right. Default is None, meaning that the value used is set automatically based on vectorizer and feature values.

- **include_styles** (*bool*) – Most styles are inline, but some are included separately in <style> tag; you can omit them by passing `include_styles=False`. Default is True.
- ****kwargs** (*dict*) – Keyword arguments. All keyword arguments are passed to concrete explain_weights... implementations.

Returns

IPython.display.HTML – The result is printed in IPython notebook as an HTML widget. If you need to display several explanations as an output of a single cell, or if you want to display it from a function then use IPython.display.display:

```
from IPython.display import display
display(eli5.show_weights(clf1))
display(eli5.show_weights(clf2))
```

show_prediction(*estimator*, *doc*, ***kwargs*)

Return an explanation of estimator prediction as an IPython.display.HTML object. Use this function to show information about classifier prediction in IPython.

show_prediction() accepts all *eli5.explain_prediction()* arguments and all *eli5.formatters.html.format_as_html()* keyword arguments, so it is possible to get explanation and customize formatting in a single call.

Parameters

- **estimator** (*object*) – Estimator instance. This argument must be positional.
- **doc** (*object*) – Example to run estimator on. Estimator makes a prediction for this example, and *show_prediction()* tries to show information about this prediction.
- **top** (*int or (int, int) tuple, optional*) – Number of features to show. When `top` is int, `top` features with a highest absolute values are shown. When it is (pos, neg) tuple, no more than `pos` positive features and no more than `neg` negative features is shown. `None` value means no limit (default).

  This argument may be supported or not, depending on estimator type.
- **target_names** (*list[str] or {'old_name': 'new_name'} dict, optional*) – Names of targets or classes. This argument can be used to provide human-readable class/target names for estimators which don't expose clss names themselves. It can be also used to rename estimator-provided classes before displaying them.

  This argument may be supported or not, depending on estimator type.
- **targets** (*list, optional*) – Order of class/target names to show. This argument can be also used to show information only for a subset of classes. It should be a list of class / target names which match either names provided by an estimator or names defined in `target_names` parameter.

  This argument may be supported or not, depending on estimator type.
- **feature_names** (*list, optional*) – A list of feature names. It allows to specify feature names when they are not provided by an estimator object.

  This argument may be supported or not, depending on estimator type.
- **horizontal_layout** (*bool*) – When True, feature weight tables are printed horizontally (left to right); when False, feature weight tables are printed vertically (top to down). Default is True.

- **highlight_spaces** (*bool or None, optional*) – Whether to highlight spaces in feature names. This is useful if you work with text and have ngram features which may include spaces at left or right. Default is None, meaning that the value used is set automatically based on vectorizer and feature values.
- **include_styles** (*bool*) – Most styles are inline, but some are included separately in <style> tag; you can omit them by passing `include_styles=False`. Default is True.
- **force_weights** (*bool*) – When True, a table with feature weights is displayed even if all features are already highlighted in text. Default is False.
- **preserve_density** (*bool or None*) – This argument currently only makes sense when used with text data and vectorizers from scikit-learn.

  If preserve_density is True, then color for longer fragments will be less intensive than for shorter fragments, so that "sum" of intensities will correspond to feature weight.

  If preserve_density is None, then it's value is chosen depending on analyzer kind: it is preserved for "char" and "char_wb" analyzers, and not preserved for "word" analyzers.

  Default is None.
- ****kwargs** (*dict*) – Keyword arguments. All keyword arguments are passed to concrete explain_prediction... implementations.

Returns

IPython.display.HTML – The result is printed in IPython notebook as an HTML widget. If you need to display several explanations as an output of a single cell, or if you want to display it from a function then use IPython.display.display:

```
from IPython.display import display
display(eli5.show_weights(clf1))
display(eli5.show_weights(clf2))
```

4.2 eli5.formatters

4.2.1 eli5.formatters.html

format_as_html(*explanation*, *include_styles=True*, *force_weights=True*, *show=('method', 'description', 'transition_features', 'targets', 'feature_importances', 'decision_tree')*, *preserve_density=None*, *highlight_spaces=None*, *horizontal_layout=True*)

Format explanation as html. Most styles are inline, but some are included separately in <style> tag, you can omit them by passing `include_styles=False` and call `format_html_styles` to render them separately (or just omit them). With `force_weights=False`, weights will not be displayed in a table for predictions where it is possible to show feature weights highlighted in the document. If `highlight_spaces` is None (default), spaces will be highlighted in feature names only if there are any spaces at the start or at the end of the feature. Setting it to True forces space highlighting, and setting it to False turns it off. If `horizontal_layout` is True (default), multiclass classifier weights are laid out horizontally.

format_hsl(*hsl_color*)

Format hsl color as css color string.

format_html_styles()

Format just the styles, use with `format_as_html(explanation, include_styles=False)`.

get_char_weights(*weighted_spans_data*, *preserve_density=None*)
Return character weights for a text document with highlighted features. If preserve_density is True, then color for longer fragments will be less intensive than for shorter fragments, so that "sum" of intensities will correspond to feature weight. If preserve_density is None, then it's value is chosen depending on analyzer kind: it is preserved for "char" and "char_wb" analyzers, and not preserved for "word" analyzers.

get_weight_range(*weights*)
Max absolute feature for pos and neg weights.

remaining_weight_color_hsl(*ws*, *weight_range*, *pos_neg*)
Color for "remaining" row. Handles a number of edge cases: if there are no weights in ws or weight_range is zero, assume the worst (most intensive positive or negative color).

weight_color_hsl(*weight*, *weight_range*, *min_lightness=0.8*)
Return HSL color components for given weight, where the max absolute weight is given by weight_range.

4.2.2 eli5.formatters.text

format_as_text(*expl*, *show=('method', 'description', 'transition_features', 'targets', 'feature_importances', 'decision_tree')*, *highlight_spaces=None*)
Format explanation as text. If `highlight_spaces` is None (default), spaces will be highlighted in feature names only if there are any spaces at the start or at the end of the feature. Setting it to True forces space highlighting, and setting it to False turns it off.

4.2.3 eli5.formatters.as_dict

format_as_dict(*explanation*)
Return a dictionary representing the explanation that can be JSON-encoded. It accepts parts of explanation (for example feature weights) as well.

4.3 eli5.lightning

eli5 supports lightning library, which contains linear classifiers with API largely compatible with scikit-learn.

explain_prediction_lightning(**args*, ***kw*)
Return an explanation of a lightning estimator predictions

explain_weights_lightning(**args*, ***kw*)
Return an explanation of a lightning estimator weights

4.4 eli5.lime

4.4.1 eli5.lime.lime

An impementation of LIME (http://arxiv.org/abs/1602.04938), an algorithm to explain predictions of black-box models.

Input:

1. a black-box classifier (or regressor - not implemented here).
2. an example to explain.

Main idea:

1. Generate a fake dataset from the example to explain: generated instances are changed versions of the example (e.g. for text it could be the same text, but with some words removed); generated labels are black-box classifier predictions for these generated examples.
2. Train a white-box classifer on these examples (e.g. a linear model).

   It helps if a white-box classifier takes in account how are examples changed in (1), e.g. uses a similar tokenization scheme.

   XXX: in the original lime code they use linear regression models trained on probability output; here we're using a classifier.
3. Explain the example through weights of this white-box classifier instead.
4. Prediction quality of a white-box classifer shows how well it approximates the black-box classifier. If the quality is low then probably explanation shouldn't be trusted (idea: choose number of examples generated at stage (1) automatically, based on a learning curve?).

   XXX: In the original lime code prediction quality is measured on the same data used for training; maybe it makes more sense to generate other examples for testing in order to prevent overfitting. Generated examples could be too close to each other though, so maybe it doesn't matter.

Tweaks:

1. Do feature selection for a white-box classifer in order to make it simpler and more explainable.
2. Weight generated examples by their distance to the original example: the further the generated example from the original example, the less it contributes to the loss function (sample_weights computed based on distance from the example to explain).

Even though the method is classifier-agnostic, it must make assumptions about the kind of features the pipeline extracts from raw data. The white-box classifer LIME uses is not required to use the same features as black-box classifier, but a mistmatch between them limits explanation quality.

get_local_pipeline_text(*text*, *predict_proba*, *n_samples=1000*, *expand_factor=10*, *random_state=None*)

Train a classifier which approximates probabilistic text classifier locally. Return (clf, vec, metrics) tuple with "easy" classifier, "easy" vectorizer, and an estimated metrics of this pipeline, i.e. how well these "easy" vectorizer/classifier approximates text classifier in neighbourhood of `text`.

4.4.2 eli5.lime.samplers

class BaseSampler

Base sampler class. Sampler is an object which generates examples similar to a given example.

sample_near(*doc*, *n_samples=1*)

Return (examples, distances) tuple with generated documents similar to a given document.

class MaskingTextSampler(*token_pattern=None*, *bow=True*, *random_state=None*)

Sampler for text data. It randomly removes words from text.

class MultivariateKernelDensitySampler(*kde=None*, *metric='euclidean'*, *fit_bandwidth=True*, *bandwidths=array([1.00000000e-06, 1.00000000e-03, 3.16227766e-03, 1.00000000e-02, 3.16227766e-02, 1.00000000e-01, 3.16227766e-01, 1.00000000e+00, 3.16227766e+00, 1.00000000e+01, 3.16227766e+01, 1.00000000e+02, 3.16227766e+02, 1.00000000e+03, 3.16227766e+03, 1.00000000e+04])*, *sigma='bandwidth'*, *n_jobs=1*, *random_state=None*)

General-purpose sampler for dense continuous data, based on multivariate kernel density estimation.

The limitation is that a single bandwidth value is used for all dimensions, i.e. bandwith matrix is a positive scalar times the identity matrix. It is a problem e.g. when features have different variances (e.g. some of them are one-hot encoded and other are continuous).

class **UnivariateKernelDensitySampler**(*kde=None*, *metric='euclidean'*, *fit_bandwidth=True*, *bandwidths=array([1.00000000e-06, 1.00000000e-03, 3.16227766e-03, 1.00000000e-02, 3.16227766e-02, 1.00000000e-01, 3.16227766e-01, 1.00000000e+00, 3.16227766e+00, 1.00000000e+01, 3.16227766e+01, 1.00000000e+02, 3.16227766e+02, 1.00000000e+03, 3.16227766e+03, 1.00000000e+04])*, *sigma='bandwidth'*, *n_jobs=1*, *random_state=None*)

General-purpose sampler for dense continuous data, based on univariate kernel density estimation. It estimates a separate probability distribution for each input dimension.

The limitation is that variable interactions are not taken in account.

Unlike KernelDensitySampler it uses different bandwidths for different dimensions; because of that it can handle one-hot encoded features somehow (make sure to at least tune the default `sigma` parameter). Also, at sampling time it replaces only random subsets of the features instead of generating totally new examples.

sample_near(*doc*, *n_samples=1*)

Sample near the document by replacing some of its features with values sampled from distribution found by KDE.

4.4.3 eli5.lime.textutils

Utilities for text generation.

cosine_similarity_vec(*num_tokens*, *num_removed_vec*)

Return cosine similarity between a binary vector with all ones of length `num_tokens` and vectors of the same length with `num_removed_vec` elements set to zero.

generate_samples(*text*, *n_samples=500*, *bow=True*, *token_pattern='(?u)\\b\\w+\\b'*, *random_state=None*)

Return `n_samples` changed versions of text (with some words removed), along with distances between the original text and a generated examples. If `bow=False`, all tokens are considered unique (i.e. token position matters).

4.5 eli5.sklearn

4.5.1 eli5.sklearn.explain_prediction

explain_prediction_linear_classifier(*clf*, *doc*, *vec=None*, *top=None*, *target_names=None*, *targets=None*, *feature_names=None*, *vectorized=False*)

Explain prediction of a linear classifier.

explain_prediction_linear_regressor(*reg*, *doc*, *vec=None*, *top=None*, *target_names=None*, *targets=None*, *feature_names=None*, *vectorized=False*)

Explain prediction of a linear regressor.

explain_prediction_sklearn(**args*, ***kw*)

Return an explanation of a scikit-learn estimator

4.5.2 eli5.sklearn.explain_weights

explain_decision_tree(*clf*, *vec=None*, *top=20*, *target_names=None*, *targets=None*, *feature_names=None*, *feature_re=None*, ***export_graphviz_kwargs*)

Return an explanation of a decision tree classifier in the following format (compatible with random forest explanations):

```
Explanation(
    estimator="<classifier repr>",
    method="<interpretation method>",
    description="<human readable description>",
    decision_tree={...tree information},
    feature_importances=[
        FeatureWeight(feature_name, importance, std_deviation),
        ...
    ]
)
```

explain_linear_classifier_weights(*clf*, *vec=None*, *top=20*, *target_names=None*, *targets=None*, *feature_names=None*, *coef_scale=None*, *feature_re=None*)

Return an explanation of a linear classifier weights in the following format:

```
Explanation(
    estimator="<classifier repr>",
    method="<interpretation method>",
    description="<human readable description>",
    targets=[
        TargetExplanation(
            target="<class name>",
            feature_weights=FeatureWeights(
                # positive weights
                pos=[
                    (feature_name, coefficient),
                    ...
                ],

                # negative weights
                neg=[
                    (feature_name, coefficient),
                    ...
                ],

                # A number of features not shown
                pos_remaining = <int>,
                neg_remaining = <int>,

                # Sum of feature weights not shown
                # pos_remaining_sum = <float>,
                # neg_remaining_sum = <float>,
            ),
        ),
        ...
    ]
)
```

To print it use utilities from eli5.formatters.

explain_linear_regressor_weights(*reg*, *vec=None*, *top=20*, *target_names=None*, *targets=None*, *feature_names=None*, *coef_scale=None*, *feature_re=None*)

Return an explanation of a linear regressor weights in the following format:

```
    Explanation(
        estimator="<regressor repr>",
        method="<interpretation method>",
        description="<human readable description>",
        targets=[
            TargetExplanation(
                target="<target name>",
                feature_weights=FeatureWeights(
                    # positive weights
                    pos=[
                        (feature_name, coefficient),
                        ...
                    ],

                    # negative weights
                    neg=[
                        (feature_name, coefficient),
                        ...
                    ],

                    # A number of features not shown
                    pos_remaining = <int>,
                    neg_remaining = <int>,

                    # Sum of feature weights not shown
                    # pos_remaining_sum = <float>,
                    # neg_remaining_sum = <float>,
                ),
            ),
            ...
        ]
    )
```

To print it use utilities from eli5.formatters.

explain_rf_feature_importance(*clf*, *vec=None*, *top=20*, *target_names=None*, *targets=None*, *feature_names=None*, *feature_re=None*)

Return an explanation of a tree-based ensemble classifier in the following format:

```
    Explanation(
        estimator="<classifier repr>",
        method="<interpretation method>",
        description="<human readable description>",
        feature_importances=[
            FeatureWeight(feature_name, importance, std_deviation),
            ...
        ]
    )
```

explain_weights_sklearn(**args*, ***kw*)

Return an explanation of an estimator

4.5.3 eli5.sklearn.unhashing

Utilities to reverse transformation done by FeatureHasher or HashingVectorizer.

class FeatureUnhasher(*hasher*, *unkn_template='FEATURE[%d]'*)

Class for recovering a mapping used by FeatureHasher.

recalculate_attributes(*force=False*)
Update all computed attributes. It is only needed if you need to access computed attributes after `patrial_fit()` was called.

class InvertableHashingVectorizer(*vec*, *unkn_template='FEATURE[%d]'*)
A wrapper for HashingVectorizer which allows to get meaningful feature names. Create it with an existing HashingVectorizer instance as an argument:

```
vec = InvertableHashingVectorizer(my_hashing_vectorizer)
```

Unlike HashingVectorizer it can be fit. During fitting InvertableHashingVectorizer learns which input terms map to which feature columns/signs; this allows to provide more meaningful *get_feature_names()*. The cost is that it is no longer stateless.

You can fit InvertableHashingVectorizer on a random sample of documents (not necessarily on the whole training and testing data), and use it to inspect an existing HashingVectorizer instance.

If several features hash to the same value, they are ordered by their frequency in documents that were used to fit the vectorizer.

`transform()` works the same as HashingVectorizer.transform.

column_signs_
Return a numpy array with expected signs of features. Values are

- +1 when all known terms which map to the column have positive sign;
- -1 when all known terms which map to the column have negative sign;
- `nan` when there are both positive and negative known terms for this column, or when there is no known term which maps to this column.

fit(*X*, *y=None*)
Extract possible terms from documents

get_feature_names(*always_signed=True*)
Return feature names. This is a best-effort function which tries to reconstruct feature names based on what it have seen so far.

HashingVectorizer uses a signed hash function. If always_signed is True, each term in feature names is prepended with its sign. If it is False, signs are only shown in case of possible collisions of different sign.

You probably want always_signed=True if you're checking unprocessed classifier coefficients, and always_signed=False if you've taken care of *column_signs_*.

4.6 eli5.sklearn_crfsuite

explain_weights_sklearn_crfsuite(*crf*, *top=20*, *target_names=None*, *targets=None*, *feature_re=None*)
Explain sklearn_crfsuite.CRF weights

filter_transition_coefs(*transition_coef*, *indices*)

```
>>> coef = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
>>> filter_transition_coefs(coef, [0])
array([[0]])
>>> filter_transition_coefs(coef, [1, 2])
array([[4, 5],
...    [7, 8]])
```

```
>>> filter_transition_coefs(coef, [2, 0])
array([[8, 6],
...    [2, 0]])
>>> filter_transition_coefs(coef, [0, 1, 2])
array([[0, 1, 2],
...    [3, 4, 5],
...    [6, 7, 8]])
```

sorted_for_ner(*crf_classes*)
Return labels sorted in a default order suitable for NER tasks:

```
>>> sorted_for_ner(['B-ORG', 'B-PER', 'O', 'I-PER'])
['O', 'B-ORG', 'B-PER', 'I-PER']
```

4.7 eli5.base

class Explanation(*estimator*, *description=None*, *error=None*, *method=None*, *is_regression=False*, *targets=None*, *feature_importances=None*, *decision_tree=None*, *highlight_spaces=None*, *transition_features=None*)
An explanation for classifier or regressor, it can either explain weights or a single prediction.

__eq__(*other*)
Automatically created by attrs.

__ge__(*other*)
Automatically created by attrs.

__gt__(*other*)
Automatically created by attrs.

__hash__()
Automatically created by attrs.

__le__(*other*)
Automatically created by attrs.

__lt__(*other*)
Automatically created by attrs.

__ne__(*other*)
Automatically created by attrs.

__repr__()
Automatically created by attrs.

class FeatureWeights(*pos*, *neg*, *pos_remaining=0*, *neg_remaining=0*)
Weights for top features, :pos: for positive and :neg: for negative, sorted by descending absolute value. Number of remaining positive and negative features are stored in :pos_remaining: and :neg_remaining: attributes.

__eq__(*other*)
Automatically created by attrs.

__ge__(*other*)
Automatically created by attrs.

__gt__(*other*)
Automatically created by attrs.

__hash__()
Automatically created by attrs.

__le__(*other*)
Automatically created by attrs.

__lt__(*other*)
Automatically created by attrs.

__ne__(*other*)
Automatically created by attrs.

__repr__()
Automatically created by attrs.

class NodeInfo(*id*, *is_leaf*, *value*, *value_ratio*, *impurity*, *samples*, *sample_ratio*, *feature_name=None*, *feature_id=None*, *threshold=None*, *left=None*, *right=None*)
A node in a binary tree. Pointers to left and right children are in :left: and :right: attributes.

__eq__(*other*)
Automatically created by attrs.

__ge__(*other*)
Automatically created by attrs.

__gt__(*other*)
Automatically created by attrs.

__hash__()
Automatically created by attrs.

__le__(*other*)
Automatically created by attrs.

__lt__(*other*)
Automatically created by attrs.

__ne__(*other*)
Automatically created by attrs.

__repr__()
Automatically created by attrs.

class TargetExplanation(*target*, *feature_weights*, *proba=None*, *score=None*, *weighted_spans=None*)
Explanation for a single target or class. Feature weights are stored in the :feature_weights: attribute, and features highlighted in text in the :weighted_spans: attribute.

__eq__(*other*)
Automatically created by attrs.

__ge__(*other*)
Automatically created by attrs.

__gt__(*other*)
Automatically created by attrs.

__hash__()
Automatically created by attrs.

__le__(*other*)
Automatically created by attrs.

__lt__(*other*)
Automatically created by attrs.

__ne__(*other*)
Automatically created by attrs.

__repr__()
Automatically created by attrs.

class TransitionFeatureWeights(*class_names*, *coef*)
Weights matrix for transition features.

__eq__(*other*)
Automatically created by attrs.

__ge__(*other*)
Automatically created by attrs.

__gt__(*other*)
Automatically created by attrs.

__hash__()
Automatically created by attrs.

__le__(*other*)
Automatically created by attrs.

__lt__(*other*)
Automatically created by attrs.

__ne__(*other*)
Automatically created by attrs.

__repr__()
Automatically created by attrs.

class TreeInfo(*criterion*, *tree*, *graphviz*)
Information about the decision tree. :criterion: is the name of the function to measure the quality of a split, :tree: holds all nodes of the tree, and :graphviz: is the tree rendered in graphviz .dot format.

__eq__(*other*)
Automatically created by attrs.

__ge__(*other*)
Automatically created by attrs.

__gt__(*other*)
Automatically created by attrs.

__hash__()
Automatically created by attrs.

__le__(*other*)
Automatically created by attrs.

__lt__(*other*)
Automatically created by attrs.

__ne__(*other*)
Automatically created by attrs.

__repr__()
Automatically created by attrs.

class WeightedSpans(*analyzer*, *document*, *weighted_spans*, *other=None*)
Features highlighted in text. :analyzer: is a type of the analyzer (for example “char” or “word”), and :document: is a pre-processed document before applying the analyzed. :weighted_spans: holds a list of spans (see above) for features found in text (span indices correspond to :document:), and :other: holds weights for features not highlighted in text.

__eq__(*other*)
Automatically created by attrs.

__ge__(*other*)
Automatically created by attrs.

__gt__(*other*)
Automatically created by attrs.

__hash__()
Automatically created by attrs.

__le__(*other*)
Automatically created by attrs.

__lt__(*other*)
Automatically created by attrs.

__ne__(*other*)
Automatically created by attrs.

__repr__()
Automatically created by attrs.

CHAPTER 5

Contributing

ELI5 uses MIT license; contributions are welcome!

- Source code: https://github.com/TeamHG-Memex/eli5
- Issue tracker: https://github.com/TeamHG-Memex/eli5/issues

ELI5 supports Python 2.7 and Python 3.4+ To run tests make sure tox Python package is installed, then run

```
tox
```

from source checkout.

We like high test coverage and mypy type annotations.

CHAPTER 6

Changelog

6.1 0.1.1 (2016-11-25)

- packaging fixes: require attrs > 16.0.0, fixed README rendering

6.2 0.1 (2016-11-24)

- HTML output;
- IPython integration;
- JSON output;
- visualization of scikit-learn text vectorizers;
- sklearn-crfsuite support;
- lightning support;
- `eli5.show_weights()` and `eli5.show_prediction()` functions;
- `eli5.explain_weights()` and `eli5.explain_prediction()` functions;
- *eli5.lime* improvements: samplers for non-text data, bug fixes, docs;
- HashingVectorizer is supported for regression tasks;
- performance improvements - feature names are lazy;
- sklearn ElasticNetCV and RidgeCV support;
- it is now possible to customize formatting output - show/hide sections, change layout;
- sklearn OneVsRestClassifier support;
- sklearn DecisionTreeClassifier visualization (text-based or svg-based);
- dropped support for scikit-learn < 0.18;
- basic mypy type annotations;
- `feature_re` argument allows to show only a subset of features;
- `target_names` argument allows to change display names of targets/classes;
- `targets` argument allows to show a subset of targets/classes and change their display order;
- documentation, more examples.

6.3 0.0.6 (2016-10-12)

- Candidate features in eli5.sklearn.InvertableHashingVectorizer are ordered by their frequency, first candidate is always positive.

6.4 0.0.5 (2016-09-27)

- HashingVectorizer support in explain_prediction;
- add an option to pass coefficient scaling array; it is useful if you want to compare coefficients for features which scale or sign is different in the input;
- bug fix: classifier weights are no longer changed by eli5 functions.

6.5 0.0.4 (2016-09-24)

- eli5.sklearn.InvertableHashingVectorizer and eli5.sklearn.FeatureUnhasher allow to recover feature names for pipelines which use HashingVectorizer or FeatureHasher;
- added support for scikit-learn linear regression models (ElasticNet, Lars, Lasso, LinearRegression, LinearSVR, Ridge, SGDRegressor);
- doc and vec arguments are swapped in explain_prediction function; vec can now be omitted if an example is already vectorized;
- fixed issue with dense feature vectors;
- all class_names arguments are renamed to target_names;
- feature name guessing is fixed for scikit-learn ensemble estimators;
- testing improvements.

6.6 0.0.3 (2016-09-21)

- support any black-box classifier using LIME (http://arxiv.org/abs/1602.04938) algorithm; text data support is built-in;
- "vectorized" argument for sklearn.explain_prediction; it allows to pass example which is already vectorized;
- allow to pass feature_names explicitly;
- support classifiers without get_feature_names method using auto-generated feature names.

6.7 0.0.2 (2016-09-19)

- 'top' argument of `explain_prediction` can be a tuple (num_positive, num_negative);
- classifier name is no longer printed by default;
- added eli5.sklearn.explain_prediction to explain individual examples;
- fixed numpy warning.

6.8 0.0.1 (2016-09-15)

Pre-release.

License is MIT.

e

eli5, 21
eli5.base, 32
eli5.formatters, 25
eli5.formatters.as_dict, 26
eli5.formatters.html, 25
eli5.formatters.text, 26
eli5.lightning, 26
eli5.lime, 26
eli5.lime.lime, 26
eli5.lime.samplers, 27
eli5.lime.textutils, 28
eli5.sklearn.explain_prediction, 28
eli5.sklearn.explain_weights, 29
eli5.sklearn.unhashing, 30
eli5.sklearn_crfsuite.explain_weights, 31

Symbols

__eq__() (Explanation method), 32
__eq__() (FeatureWeights method), 32
__eq__() (NodeInfo method), 33
__eq__() (TargetExplanation method), 33
__eq__() (TransitionFeatureWeights method), 34
__eq__() (TreeInfo method), 34
__eq__() (WeightedSpans method), 34
__ge__() (Explanation method), 32
__ge__() (FeatureWeights method), 32
__ge__() (NodeInfo method), 33
__ge__() (TargetExplanation method), 33
__ge__() (TransitionFeatureWeights method), 34
__ge__() (TreeInfo method), 34
__ge__() (WeightedSpans method), 35
__gt__() (Explanation method), 32
__gt__() (FeatureWeights method), 32
__gt__() (NodeInfo method), 33
__gt__() (TargetExplanation method), 33
__gt__() (TransitionFeatureWeights method), 34
__gt__() (TreeInfo method), 34
__gt__() (WeightedSpans method), 35
__hash__() (Explanation method), 32
__hash__() (FeatureWeights method), 32
__hash__() (NodeInfo method), 33
__hash__() (TargetExplanation method), 33
__hash__() (TransitionFeatureWeights method), 34
__hash__() (TreeInfo method), 34
__hash__() (WeightedSpans method), 35
__le__() (Explanation method), 32
__le__() (FeatureWeights method), 33
__le__() (NodeInfo method), 33
__le__() (TargetExplanation method), 33
__le__() (TransitionFeatureWeights method), 34
__le__() (TreeInfo method), 34
__le__() (WeightedSpans method), 35
__lt__() (Explanation method), 32
__lt__() (FeatureWeights method), 33
__lt__() (NodeInfo method), 33
__lt__() (TargetExplanation method), 33
__lt__() (TransitionFeatureWeights method), 34
__lt__() (TreeInfo method), 34
__lt__() (WeightedSpans method), 35
__ne__() (Explanation method), 32
__ne__() (FeatureWeights method), 33
__ne__() (NodeInfo method), 33
__ne__() (TargetExplanation method), 33
__ne__() (TransitionFeatureWeights method), 34
__ne__() (TreeInfo method), 34
__ne__() (WeightedSpans method), 35
__repr__() (Explanation method), 32
__repr__() (FeatureWeights method), 33
__repr__() (NodeInfo method), 33
__repr__() (TargetExplanation method), 33
__repr__() (TransitionFeatureWeights method), 34
__repr__() (TreeInfo method), 34
__repr__() (WeightedSpans method), 35

B

BaseSampler (class in eli5.lime.samplers), 27

C

column_signs_ (InvertableHashingVectorizer attribute), 31
cosine_similarity_vec() (in module eli5.lime.textutils), 28

E

eli5 (module), 21
eli5.base (module), 32
eli5.formatters (module), 25
eli5.formatters.as_dict (module), 26
eli5.formatters.html (module), 25
eli5.formatters.text (module), 26
eli5.lightning (module), 26
eli5.lime (module), 26
eli5.lime.lime (module), 26
eli5.lime.samplers (module), 27
eli5.lime.textutils (module), 28
eli5.sklearn.explain_prediction (module), 28
eli5.sklearn.explain_weights (module), 29

eli5.sklearn.unhashing (module), 30
eli5.sklearn_crfsuite.explain_weights (module), 31
explain_decision_tree() (in module eli5.sklearn.explain_weights), 29
explain_linear_classifier_weights() (in module eli5.sklearn.explain_weights), 29
explain_linear_regressor_weights() (in module eli5.sklearn.explain_weights), 29
explain_prediction() (in module eli5), 22
explain_prediction_lightning() (in module eli5.lightning), 26
explain_prediction_linear_classifier() (in module eli5.sklearn.explain_prediction), 28
explain_prediction_linear_regressor() (in module eli5.sklearn.explain_prediction), 28
explain_prediction_sklearn() (in module eli5.sklearn.explain_prediction), 28
explain_rf_feature_importance() (in module eli5.sklearn.explain_weights), 30
explain_weights() (in module eli5), 21
explain_weights_lightning() (in module eli5.lightning), 26
explain_weights_sklearn() (in module eli5.sklearn.explain_weights), 30
explain_weights_sklearn_crfsuite() (in module eli5.sklearn_crfsuite.explain_weights), 31
Explanation (class in eli5.base), 32

F

FeatureUnhasher (class in eli5.sklearn.unhashing), 30
FeatureWeights (class in eli5.base), 32
filter_transition_coefs() (in module eli5.sklearn_crfsuite.explain_weights), 31
fit() (InvertableHashingVectorizer method), 31
format_as_dict() (in module eli5.formatters.as_dict), 26
format_as_html() (in module eli5.formatters.html), 25
format_as_text() (in module eli5.formatters.text), 26
format_hsl() (in module eli5.formatters.html), 25
format_html_styles() (in module eli5.formatters.html), 25

G

generate_samples() (in module eli5.lime.textutils), 28
get_char_weights() (in module eli5.formatters.html), 25
get_feature_names() (InvertableHashingVectorizer method), 31
get_local_pipeline_text() (in module eli5.lime.lime), 27
get_weight_range() (in module eli5.formatters.html), 26

I

InvertableHashingVectorizer (class in eli5.sklearn.unhashing), 31

M

MaskingTextSampler (class in eli5.lime.samplers), 27
MultivariateKernelDensitySampler (class in eli5.lime.samplers), 27

N

NodeInfo (class in eli5.base), 33

R

recalculate_attributes() (FeatureUnhasher method), 30
remaining_weight_color_hsl() (in module eli5.formatters.html), 26

S

sample_near() (BaseSampler method), 27
sample_near() (UnivariateKernelDensitySampler method), 28
show_prediction() (in module eli5), 24
show_weights() (in module eli5), 22
sorted_for_ner() (in module eli5.sklearn_crfsuite.explain_weights), 32

T

TargetExplanation (class in eli5.base), 33
TransitionFeatureWeights (class in eli5.base), 34
TreeInfo (class in eli5.base), 34

U

UnivariateKernelDensitySampler (class in eli5.lime.samplers), 28

W

weight_color_hsl() (in module eli5.formatters.html), 26
WeightedSpans (class in eli5.base), 34