ELI5 Documentation

Release 0.6.1

Mikhail Korobov, Konstantin Lopuhin

May 10, 2017

Contents

1 Overview **3**
- 1.1 Installation . . . 3
- 1.2 Features . . . 3
- 1.3 Basic Usage . . . 4
- 1.4 Why? . . . 4
- 1.5 Architecture . . . 5

2 Tutorials **7**
- 2.1 Debugging scikit-learn text classification pipeline . . . 7
- 2.2 TextExplainer: debugging black-box text classifiers . . . 14
- 2.3 Explaining XGBoost predictions on the Titanic dataset . . . 22
- 2.4 Named Entity Recognition using sklearn-crfsuite . . . 27

3 Supported Libraries **33**
- 3.1 scikit-learn . . . 33
- 3.2 XGBoost . . . 38
- 3.3 LightGBM . . . 38
- 3.4 lightning . . . 39
- 3.5 sklearn-crfsuite . . . 40

4 LIME **41**
- 4.1 Algorithm . . . 41
- 4.2 eli5.lime . . . 41
- 4.3 Caveats . . . 41
- 4.4 Alternative implementations . . . 42

5 API **43**
- 5.1 ELI5 top-level API . . . 43
- 5.2 eli5.formatters . . . 49
- 5.3 eli5.lightning . . . 50
- 5.4 eli5.lime . . . 50
- 5.5 eli5.sklearn . . . 54
- 5.6 eli5.sklearn_crfsuite . . . 58
- 5.7 eli5.xgboost . . . 58
- 5.8 eli5.lightgbm . . . 59
- 5.9 eli5.base . . . 60

6 Contributing **61**

7 Changelog **63**
7.1 0.6.1 (2017-05-10) . . . 63
7.2 0.6 (2017-05-03) . . . 63
7.3 0.5 (2017-04-27) . . . 63
7.4 0.4.2 (2017-03-03) . . . 64
7.5 0.4.1 (2017-01-25) . . . 64
7.6 0.4 (2017-01-20) . . . 64
7.7 0.3.1 (2017-01-16) . . . 64
7.8 0.3 (2017-01-13) . . . 64
7.9 0.2 (2016-12-03) . . . 65
7.10 0.1.1 (2016-11-25) . . . 65
7.11 0.1 (2016-11-24) . . . 65
7.12 0.0.6 (2016-10-12) . . . 66
7.13 0.0.5 (2016-09-27) . . . 66
7.14 0.0.4 (2016-09-24) . . . 66
7.15 0.0.3 (2016-09-21) . . . 66
7.16 0.0.2 (2016-09-19) . . . 67
7.17 0.0.1 (2016-09-15) . . . 67

Python Module Index **69**

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

Installation

ELI5 works in Python 2.7 and Python 3.4+. Currently it requires scikit-learn 0.18+. You can install ELI5 using pip:

```
pip install eli5
```

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 and explain predictions of decision trees and tree-based ensembles.

  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.
- *XGBoost* - show feature importances and explain predictions of XGBClassifier and XGBRegressor.
- *LightGBM* - show feature importances and explain predictions of LGBMClassifier and LGBMRegressor.
- *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 *TextExplainer* which allows to explain predictions of any text classifier using *LIME* algorithm (Ribeiro et al., 2016). There are utilities for using LIME with non-text data and arbitrary black-box classifiers as well, but 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.

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 something like this in the IPython Notebook:

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

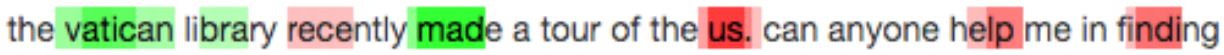
and get an explanation like this:

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

Note: Depending on an estimator, you may need to pass additional parameters to get readable results - e.g. a vectorizer used to prepare features for a classifier, or a list of feature names.

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 specific to your framework in the *Supported Libraries* section.

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.

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.

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.

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. 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.

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!

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.

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 can be run as an IPython notebook.

TextExplainer: debugging black-box text classifiers

While eli5 supports many classifiers and preprocessing methods, it can't support them all.

If a library is not supported by eli5 directly, or the text processing pipeline is too complex for eli5, eli5 can still help - it provides an implementation of LIME (Ribeiro et al., 2016) algorithm which allows to explain predictions of arbitrary classifiers, including text classifiers. `eli5.lime` can also help when it is hard to get exact mapping between model coefficients and text features, e.g. if there is dimension reduction involved.

Example problem: LSA+SVM for 20 Newsgroups dataset

Let's load "20 Newsgroups" dataset and create a text processing pipeline which is hard to debug using conventional methods: SVM with RBF kernel trained on LSA features.

```
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,
    remove=('headers', 'footers'),
)
```

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

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.decomposition import TruncatedSVD
from sklearn.pipeline import Pipeline, make_pipeline

vec = TfidfVectorizer(min_df=3, stop_words='english',
                      ngram_range=(1, 2))
svd = TruncatedSVD(n_components=100, n_iter=7, random_state=42)
lsa = make_pipeline(vec, svd)

clf = SVC(C=150, gamma=2e-2, probability=True)
pipe = make_pipeline(lsa, clf)
pipe.fit(twenty_train.data, twenty_train.target)
pipe.score(twenty_test.data, twenty_test.target)
```

```
0.89014647137150471
```

The dimension of the input documents is reduced to 100, and then a kernel SVM is used to classify the documents.

This is what the pipeline returns for a document - it is pretty sure the first message in test data belongs to sci.med:

```
def print_prediction(doc):
    y_pred = pipe.predict_proba([doc])[0]
    for target, prob in zip(twenty_train.target_names, y_pred):
        print("{:.3f} {}".format(prob, target))

doc = twenty_test.data[0]
print_prediction(doc)
```

```
0.001 alt.atheism
0.001 comp.graphics
0.995 sci.med
0.004 soc.religion.christian
```

TextExplainer

Such pipelines are not supported by eli5 directly, but one can use `eli5.lime.TextExplainer` to debug the prediction - to check what was important in the document to make this decision.

Create a *`TextExplainer`* instance, then pass the document to explain and a black-box classifier (a function which returns probabilities) to the *`fit()`* method, then check the explanation:

```
import eli5
from eli5.lime import TextExplainer

te = TextExplainer(random_state=42)
te.fit(doc, pipe.predict_proba)
te.show_prediction(target_names=twenty_train.target_names)
```

Why it works

Explanation makes sense - we expect reasonable classifier to take highlighted words in account. But how can we be sure this is how the pipeline works, not just a nice-looking lie? A simple sanity check is to remove or change the highlighted words, to confirm that they change the outcome:

```
import re
doc2 = re.sub(r'(recall|kidney|stones|medication|pain|tech)', '', doc, flags=re.I)
print_prediction(doc2)
```

```
0.065 alt.atheism
0.145 comp.graphics
0.376 sci.med
0.414 soc.religion.christian
```

Predicted probabilities changed a lot indeed.

And in fact, *TextExplainer* did something similar to get the explanation. *TextExplainer* generated a lot of texts similar to the document (by removing some of the words), and then trained a white-box classifier which predicts the output of the black-box classifier (not the true labels!). The explanation we saw is for this white-box classifier.

This approach follows the LIME algorithm; for text data the algorithm is actually pretty straightforward:

1. generate distorted versions of the text;
2. predict probabilities for these distorted texts using the black-box classifier;
3. train another classifier (one of those eli5 supports) which tries to predict output of a black-box classifier on these texts.

The algorithm works because even though it could be hard or impossible to approximate a black-box classifier globally (for every possible text), approximating it in a small neighbourhood near a given text often works well, even with simple white-box classifiers.

Generated samples (distorted texts) are available in `samples_` attribute:

```
print(te.samples_[0])
```

```
As    my   kidney ,  isn' any
  can        .

Either they ,     be    ,
to   .

   ,  - tech  to mention  ' had kidney
 and ,     .
```

By default *TextExplainer* generates 5000 distorted texts (use `n_samples` argument to change the amount):

```
len(te.samples_)
```

```
5000
```

Trained white-box classifier and vectorizer are available as `vec_` and `clf_` attributes:

```
te.vec_, te.clf_
```

```
(CountVectorizer(analyzer='word', binary=False, decode_error='strict',
         dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
         lowercase=True, max_df=1.0, max_features=None, min_df=1,
         ngram_range=(1, 2), preprocessor=None, stop_words=None,
         strip_accents=None, token_pattern='(?u)\b\w+\b', tokenizer=None,
         vocabulary=None),
 SGDClassifier(alpha=0.001, average=False, class_weight=None, epsilon=0.1,
        eta0=0.0, fit_intercept=True, l1_ratio=0.15,
        learning_rate='optimal', loss='log', n_iter=5, n_jobs=1,
        penalty='elasticnet', power_t=0.5,
        random_state=<mtrand.RandomState object at 0x10e1dcf78>,
        shuffle=True, verbose=0, warm_start=False))
```

Should we trust the explanation?

Ok, this sounds fine, but how can we be sure that this simple text classification pipeline approximated the black-box classifier well?

One way to do that is to check the quality on a held-out dataset (which is also generated). *TextExplainer* does that by default and stores metrics in `metrics_` attribute:

```
te.metrics_
```

```
{'mean_KL_divergence': 0.020120624088861134, 'score': 0.98625304704899297}
```

- 'score' is an accuracy score weighted by cosine distance between generated sample and the original document (i.e. texts which are closer to the example are more important). Accuracy shows how good are 'top 1' predictions.
- 'mean_KL_divergence' is a mean Kullback–Leibler divergence for all target classes; it is also weighted by distance. KL divergence shows how well are probabilities approximated; 0.0 means a perfect match.

In this example both accuracy and KL divergence are good; it means our white-box classifier usually assigns the same labels as the black-box classifier on the dataset we generated, and its predicted probabilities are close to those predicted by our LSA+SVM pipeline. So it is likely (though not guaranteed, we'll discuss it later) that the explanation is correct and can be trusted.

When working with LIME (e.g. via *TextExplainer*) it is always a good idea to check these scores. If they are not good then you can tell that something is not right.

Let's make it fail

By default *TextExplainer* uses a very basic text processing pipeline: Logistic Regression trained on bag-of-words and bag-of-bigrams features (see `te.clf_` and `te.vec_` attributes). It limits a set of black-box classifiers it can explain: because the text is seen as "bag of words/ngrams", the default white-box pipeline can't distinguish e.g. between the same word in the beginning of the document and in the end of the document. Bigrams help to alleviate the problem in practice, but not completely.

Black-box classifiers which use features like "text length" (not directly related to tokens) can be also hard to approximate using the default bag-of-words/ngrams model.

This kind of failure is usually detectable though - scores (accuracy and KL divergence) will be low. Let's check it on a completely synthetic example - a black-box classifier which assigns a class based on oddity of document length and on a presence of 'medication' word.

```
import numpy as np

def predict_proba_len(docs):
    # nasty predict_proba - the result is based on document length,
    # and also on a presence of "medication"
    proba = [
        [0, 0, 1.0, 0] if len(doc) % 2 or 'medication' in doc else [1.0, 0, 0, 0]
        for doc in docs
    ]
    return np.array(proba)

te3 = TextExplainer().fit(doc, predict_proba_len)
te3.show_prediction(target_names=twenty_train.target_names)
```

TextExplainer correctly figured out that 'medication' is important, but failed to account for "len(doc) % 2" condition, so the explanation is incomplete. We can detect this failure by looking at metrics - they are low:

```
te3.metrics_
```

```
{'mean_KL_divergence': 0.3312922355257879, 'score': 0.79050673156810314}
```

If (a big if...) we suspect that the fact document length is even or odd is important, it is possible to customize *TextExplainer* to check this hypothesis.

To do that, we need to create a vectorizer which returns both "is odd" feature and bag-of-words features, and pass this vectorizer to *TextExplainer*. This vectorizer should follow scikit-learn API. The easiest way is to use `FeatureUnion` - just make sure all transformers joined by `FeatureUnion` have `get_feature_names()` methods.

```
from sklearn.pipeline import make_union
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.base import TransformerMixin

class DocLength(TransformerMixin):
    def fit(self, X, y=None):  # some boilerplate
        return self

    def transform(self, X):
        return [
            # note that we needed both positive and negative
            # feature - otherwise for linear model there won't
            # be a feature to show in a half of the cases
            [len(doc) % 2, not len(doc) % 2]
            for doc in X
        ]

    def get_feature_names(self):
        return ['is_odd', 'is_even']

vec = make_union(DocLength(), CountVectorizer(ngram_range=(1,2)))
te4 = TextExplainer(vec=vec).fit(doc[:-1], predict_proba_len)

print(te4.metrics_)
te4.explain_prediction(target_names=twenty_train.target_names)
```

```
{'mean_KL_divergence': 0.024826114773734968, 'score': 1.0}
```

Much better! It was a toy example, but the idea stands - if you think something could be important, add it to the mix as a feature for *TextExplainer*.

Let's make it fail, again

Another possible issue is the dataset generation method. Not only feature extraction should be powerful enough, but auto-generated texts also should be diverse enough.

TextExplainer removes random words by default, so by default it can't e.g. provide a good explanation for a black-box classifier which works on character level. Let's try to use *TextExplainer* to explain a classifier which uses char ngrams as features:

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

vec_char = HashingVectorizer(analyzer='char_wb', ngram_range=(4,5))
clf_char = SGDClassifier(loss='log')

pipe_char = make_pipeline(vec_char, clf_char)
pipe_char.fit(twenty_train.data, twenty_train.target)
pipe_char.score(twenty_test.data, twenty_test.target)
```

```
0.88082556591211714
```

This pipeline is supported by eli5 directly, so in practice there is no need to use *TextExplainer* for it. We're using this pipeline as an example - it is possible check the "true" explanation first, without using *TextExplainer*, and then compare the results with *TextExplainer* results.

```
eli5.show_prediction(clf_char, doc, vec=vec_char,
                     targets=['sci.med'], target_names=twenty_train.target_names)
```

TextExplainer produces a different result:

```
te = TextExplainer(random_state=42).fit(doc, pipe_char.predict_proba)
print(te.metrics_)
te.show_prediction(targets=['sci.med'], target_names=twenty_train.target_names)
```

```
{'mean_KL_divergence': 0.020247299052285436, 'score': 0.92434669226497945}
```

Scores look OK but not great; the explanation kind of makes sense on a first sight, but we know that the classifier works in a different way.

To explain such black-box classifiers we need to change both dataset generation method (change/remove individual characters, not only words) and feature extraction method (e.g. use char ngrams instead of words and word ngrams).

TextExplainer has an option (`char_based=True`) to use char-based sampling and char-based classifier. If this makes a more powerful explanation engine why not always use it?

```
te = TextExplainer(char_based=True, random_state=42)
te.fit(doc, pipe_char.predict_proba)
print(te.metrics_)
te.show_prediction(targets=['sci.med'], target_names=twenty_train.target_names)
```

```
{'mean_KL_divergence': 0.22136004391576117, 'score': 0.5566945067868848 1}
```

Hm, the result look worse. *TextExplainer* detected correctly that only the first part of word "medication" is important, but the result is noisy overall, and scores are bad. Let's try it with more samples:

```
te = TextExplainer(char_based=True, n_samples=50000, random_state=42)
te.fit(doc, pipe_char.predict_proba)
print(te.metrics_)
te.show_prediction(targets=['sci.med'], target_names=twenty_train.target_names)
```

```
{'mean_KL_divergence': 0.060019833958355841, 'score': 0.86048000626542609}
```

It is getting closer, but still not there yet. The problem is that it is much more resource intensive - you need a lot more samples to get non-noisy results. Here explaining a single example took more time than training the original pipeline.

Generally speaking, to do an efficient explanation we should make some assumptions about black-box classifier, such as:

1. it uses words as features and doesn't take word position in account;
2. it uses words as features and takes word positions in account;
3. it uses words ngrams as features;
4. it uses char ngrams as features, positions don't matter (i.e. an ngram means the same everywhere);
5. it uses arbitrary attention over the text characters, i.e. every part of text could be potentionally important for a classifier on its own;
6. it is important to have a particular token at a particular position, e.g. "third token is X", and if we delete 2nd token then prediction changes not because 2nd token changed, but because 3rd token is shifted.

Depending on assumptions we should choose both dataset generation method and a white-box classifier. There is a tradeoff between generality and speed.

Simple bag-of-words assumptions allow for fast sample generation, and just a few hundreds of samples could be required to get an OK quality if the assumption is correct. But such generation methods / models will fail to explain a more complex classifier properly (they could still provide an explanation which is useful in practice though).

On the other hand, allowing for each character to be important is a more powerful method, but it can require a lot of samples (maybe hundreds thousands) and a lot of CPU time to get non-noisy results.

What's bad about this kind of failure (wrong assumption about the black-box pipeline) is that it could be impossible to detect the failure by looking at the scores. Scores could be high because generated dataset is not diverse enough, not because our approximation is good.

The takeaway is that it is important to understand the "lenses" you're looking through when using LIME to explain a prediction.

Customizing TextExplainer: sampling

TextExplainer uses *MaskingTextSampler* or *MaskingTextSamplers* instances to generate texts to train on. *MaskingTextSampler* is the main text generation class; *MaskingTextSamplers* provides a way to combine multiple samplers in a single object with the same interface.

A custom sampler instance can be passed to *TextExplainer* if we want to experiment with sampling. For example, let's try a sampler which replaces no more than 3 characters in the text (default is to replace a random number of characters):

```
from eli5.lime.samplers import MaskingTextSampler
sampler = MaskingTextSampler(
    # Regex to split text into tokens.
    # "." means any single character is a token, i.e.
    # we work on chars.
    token_pattern='.',

    # replace no more than 3 tokens
    max_replace=3,

    # by default all tokens are replaced;
    # replace only a token at a given position.
    bow=False,
)
samples, similarity = sampler.sample_near(doc)
print(samples[0])
```

```
As I recal from my bout with kidney stones, there isn't any
medication that can do anything about them except relieve the ain.

Either thy pass, or they have to be broken up with sound, or they have
to be extracted surgically.

When I was in, the X-ray tech happened to mention that she'd had kidney
stones and children, and the childbirth hurt less.
```

```
te = TextExplainer(char_based=True, sampler=sampler, random_state=42)
te.fit(doc, pipe_char.predict_proba)
print(te.metrics_)
te.show_prediction(targets=['sci.med'], target_names=twenty_train.target_names)
```

```
{'mean_KL_divergence': 0.71042368337755823, 'score': 0.99933430578588944}
```

Note that accuracy score is perfect, but KL divergence is bad. It means this sampler was not very useful: most generated texts were "easy" in sense that most (or all?) of them should be still classified as `sci.med`, so it was easy to get a good accuracy. But because generated texts were not diverse enough classifier haven't learned anything useful; it's having a hard time predicting the probability output of the black-box pipeline on a held-out dataset.

By default *`TextExplainer`* uses a mix of several sampling strategies which seems to work OK for token-based explanations. But a good sampling strategy which works for many real-world tasks could be a research topic on itself. If you've got some experience with it we'd love to hear from you - please share your findings in eli5 issue tracker (https://github.com/TeamHG-Memex/eli5/issues)!

Customizing TextExplainer: classifier

In one of the previous examples we already changed the vectorizer TextExplainer uses (to take additional features in account). It is also possible to change the white-box classifier - for example, use a small decision tree:

```
from sklearn.tree import DecisionTreeClassifier

te5 = TextExplainer(clf=DecisionTreeClassifier(max_depth=2), random_state=0)
te5.fit(doc, pipe.predict_proba)
print(te5.metrics_)
te5.show_weights()
```

```
{'mean_KL_divergence': 0.037836554598348969, 'score': 0.9838155527960798}
```

How to read it: "kidney <= 0.5" means "word 'kidney' is not in the document" (we're explaining the orginal LDA+SVM pipeline again).

So according to this tree if "kidney" is not in the document and "pain" is not in the document then the probability of a document belonging to `sci.med` drops to `0.65`. If at least one of these words remain `sci.med` probability stays `0.9+`.

```
print("both words removed::")
print_prediction(re.sub(r"(kidney|pain)", "", doc, flags=re.I))
print("\nonly 'pain' removed:")
print_prediction(re.sub(r"pain", "", doc, flags=re.I))
```

```
both words removed::
0.013 alt.atheism
0.022 comp.graphics
0.894 sci.med
0.072 soc.religion.christian

only 'pain' removed:
0.002 alt.atheism
0.004 comp.graphics
0.979 sci.med
0.015 soc.religion.christian
```

As expected, after removing both words probability of `sci.med` decreased, though not as much as our simple decision tree predicted (to 0.9 instead of 0.64). Removing `pain` provided exactly the same effect as predicted - probability of `sci.med` became `0.98`.

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

Explaining XGBoost predictions on the Titanic dataset

This tutorial will show you how to analyze predictions of an XGBoost classifier (regression for XGBoost and most scikit-learn tree ensembles are also supported by eli5). We will use Titanic dataset, which is small and has not too many features, but is still interesting enough.

We are using XGBoost 0.6a2 and data downloaded from https://www.kaggle.com/c/titanic/data (it is also bundled in the eli5 repo: https://github.com/TeamHG-Memex/eli5/blob/master/notebooks/titanic-train.csv).

1. Training data

Let's start by loading the data:

```
import csv
import numpy as np

with open('titanic-train.csv', 'rt') as f:
    data = list(csv.DictReader(f))
data[:1]
```

```
[{'Age': '22',
  'Cabin': '',
  'Embarked': 'S',
  'Fare': '7.25',
  'Name': 'Braund, Mr. Owen Harris',
  'Parch': '0',
  'PassengerId': '1',
  'Pclass': '3',
  'Sex': 'male',
  'SibSp': '1',
  'Survived': '0',
  'Ticket': 'A/5 21171'}]
```

Variable descriptions:

- **Age:** Age
- **Cabin:** Cabin
- **Embarked:** Port of Embarkation (C = Cherbourg; Q = Queenstown; S = Southampton)
- **Fare:** Passenger Fare
- **Name:** Name
- **Parch:** Number of Parents/Children Aboard
- **Pclass:** Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
- **Sex:** Sex
- **Sibsp:** Number of Siblings/Spouses Aboard
- **Survived:** Survival (0 = No; 1 = Yes)
- **Ticket:** Ticket Number

Next, shuffle data and separate features from what we are trying to predict: survival.

```
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split

_all_xs = [{k: v for k, v in row.items() if k != 'Survived'} for row in data]
_all_ys = np.array([int(row['Survived']) for row in data])

all_xs, all_ys = shuffle(_all_xs, _all_ys, random_state=0)
train_xs, valid_xs, train_ys, valid_ys = train_test_split(
    all_xs, all_ys, test_size=0.25, random_state=0)
print('{} items total, {:.1%} true'.format(len(all_xs), np.mean(all_ys)))
```

```
891 items total, 38.4% true
```

We do just minimal preprocessing: convert obviously contiuous *Age* and *Fare* variables to floats, and *SibSp*, *Parch* to integers. Missing *Age* values are removed.

```
for x in all_xs:
    if x['Age']:
        x['Age'] = float(x['Age'])
    else:
        x.pop('Age')
    x['Fare'] = float(x['Fare'])
```

```
    x['SibSp'] = int(x['SibSp'])
    x['Parch'] = int(x['Parch'])
```

2. Simple XGBoost classifier

Let's first build a very simple classifier with xbgoost.XGBClassifier and sklearn.feature_extraction.DictVectorizer, and check its accuracy with 10-fold cross-validation:

```
import warnings
# xgboost <= 0.6a2 shows a warning when used with scikit-learn 0.18+
warnings.filterwarnings('ignore', category=DeprecationWarning)
from xgboost import XGBClassifier
from sklearn.feature_extraction import DictVectorizer
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score

class CSCTransformer:
    def transform(self, xs):
        # work around https://github.com/dmlc/xgboost/issues/1238#issuecomment-
↪243872543
        return xs.tocsc()
    def fit(self, *args):
        return self

clf = XGBClassifier()
vec = DictVectorizer()
pipeline = make_pipeline(vec, CSCTransformer(), clf)

def evaluate(_clf):
    scores = cross_val_score(_clf, all_xs, all_ys, scoring='accuracy', cv=10)
    print('Accuracy: {:.3f} ± {:.3f}'.format(np.mean(scores), 2 * np.std(scores)))
    _clf.fit(train_xs, train_ys)  # so that parts of the original pipeline are fitted

evaluate(pipeline)
```

```
Accuracy: 0.823 ± 0.071
```

There is one tricky bit about the code above: XGBClassifier in xgboost 0.6a2 has some issues with sparse data. One way to solve them is to convert a sparse matrix to CSC format, so we add a `CSCTransformer` to the pipelne. One may be templed to just pass `dense=True` to `DictVectorizer`: after all, in this case the matrixes are small. But this is not a great solution, because we will loose the ability to distinguish features that are missing and features that have zero value.

3. Explaining weights

In order to calculate a prediction, XGBoost sums predictions of all its trees. The number of trees is controlled by `n_estimators` argument and is 100 by default. Each tree is not a great predictor on it's own, but by summing across all trees, XGBoost is able to provide a robust estimate in many cases. Here is one of the trees:

```
booster = clf.booster()
original_feature_names = booster.feature_names
booster.feature_names = vec.get_feature_names()
print(booster.get_dump()[0])
```

```
# recover original feature names
booster.feature_names = original_feature_names
```

```
0:[Sex=female<-9.53674e-07] yes=1,no=2,missing=1
    1:[Age<13] yes=3,no=4,missing=4
            3:[SibSp<2] yes=7,no=8,missing=7
                    7:leaf=0.145455
                    8:leaf=-0.125
            4:[Fare<26.2687] yes=9,no=10,missing=9
                    9:leaf=-0.151515
                    10:leaf=-0.0727273
    2:[Pclass=3<-9.53674e-07] yes=5,no=6,missing=5
            5:[Fare<12.175] yes=11,no=12,missing=12
                    11:leaf=0.05
                    12:leaf=0.175194
            6:[Fare<24.8083] yes=13,no=14,missing=14
                    13:leaf=0.0365591
                    14:leaf=-0.152
```

We see that this tree checks *Sex*, *Age*, *Pclass*, *Fare* and *SibSp* features. `leaf` gives the decision of a single tree, and they are summed over all trees in the ensemble.

Let's check feature importances with *eli5.show_weights()*:

```
from eli5 import show_weights
show_weights(clf, vec=vec)
```

There are several different ways to calculate feature importances. By default, "gain" is used, that is the average gain of the feature when it is used in trees. Other types are "weight" - the number of times a feature is used to split the data, and "cover" - the average coverage of the feature. You can pass it with `importance_type` argument.

Now we know that two most important features are *Sex=female* and *Pclass=3*, but we still don't know how XGBoost decides what prediction to make based on their values.

4. Explaining predictions

To get a better idea of how our classifier works, let's examine individual predictions with *eli5.show_prediction()*:

```
from eli5 import show_prediction
show_prediction(clf, valid_xs[1], vec=vec, show_feature_values=True)
```

Weight means how much each feature contributed to the final prediction across all trees. The idea for weight calculation is described in http://blog.datadive.net/interpreting-random-forests/; eli5 provides an independent implementation of this algorithm for XGBoost and most scikit-learn tree ensembles.

Here we see that classifier thinks it's good to be a female, but bad to travel third class. Some features have "Missing" as value (we are passing `show_feature_values=True` to view the values): that means that the feature was missing, so in this case it's good to not have embarked in Southampton. This is where our decision to go with sparse matrices comes handy - we still see that *Parch* is zero, not missing.

It's possible to show only features that are present using `feature_filter` argument: it's a function that accepts feature name and value, and returns True value for features that should be shown:

```
no_missing = lambda feature_name, feature_value: not np.isnan(feature_value)
show_prediction(clf, valid_xs[1], vec=vec, show_feature_values=True, feature_
↪filter=no_missing)
```

5. Adding text features

Right now we treat *Name* field as categorical, like other text features. But in this dataset each name is unique, so XGBoost does not use this feature at all, because it's such a poor discriminator: it's absent from the weights table in section 3.

But *Name* still might contain some useful information. We don't want to guess how to best pre-process it and what features to extract, so let's use the most general character ngram vectorizer:

```
from sklearn.pipeline import FeatureUnion
from sklearn.feature_extraction.text import CountVectorizer

vec2 = FeatureUnion([
    ('Name', CountVectorizer(
        analyzer='char_wb',
        ngram_range=(3, 4),
        preprocessor=lambda x: x['Name'],
        max_features=100,
    )),
    ('All', DictVectorizer()),
])
clf2 = XGBClassifier()
pipeline2 = make_pipeline(vec2, CSCTransformer(), clf2)
evaluate(pipeline2)
```

```
Accuracy: 0.839 ± 0.081
```

In this case the pipeline is more complex, we slightly improved our result, but the improvement is not significant. Let's look at feature importances:

```
show_weights(clf2, vec=vec2)
```

We see that now there is a lot of features that come from the *Name* field (in fact, a classifier based on *Name* alone gives about 0.79 accuracy). Name features listed in this way are not very informative, they make more sense when we check out predictions. We hide missing features here because there is a lot of missing features in text, but they are not very interesting:

```
from IPython.display import display

for idx in [4, 5, 7, 37, 81]:
    display(show_prediction(clf2, valid_xs[idx], vec=vec2,
                            show_feature_values=True, feature_filter=no_missing))
```

Text features from the *Name* field are highlighted directly in text, and the sum of weights is shown in the weights table as "Name: Highlighted in text (sum)".

Looks like name classifier tried to infer both gender and status from the title: "Mr." is bad because women are saved first, and it's better to be "Mrs." (married) than "Miss.". Also name classifier is trying to pick some parts of names and surnames, especially endings, perhaps as a proxy for social status. It's especially bad to be "Mary" if you are from the third class.

Note: This tutorial can be run as an IPython notebook.

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
```

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. 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:]': '('}
```

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);
```

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)
```

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

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

If we regularize CRF more, we can expect that only features which are generic will remain, and memoized tokens will go. With L1 regularization (c1 parameter) coefficients of most features should be driven to zero. Let's check what effect does regularization have on CRF weights:

```
crf = sklearn_crfsuite.CRF(
    algorithm='lbfgs',
    c1=200,
    c2=0.1,
    max_iterations=20,
    all_possible_transitions=False,
)
crf.fit(X_train, y_train)
eli5.show_weights(crf, top=30)
```

As you can see, memoized tokens are mostly gone and model now relies on word shapes and POS tags. There is only a few non-zero features remaining. In our example the change probably made the quality worse, but that's a separate question.

Let's focus on transition weights. We can expect that O -> I-ENTIRY transitions to have large negative weights because they are impossible. But these transitions have zero weights, not negative weights, both in heavily regularized model and in our initial model. Something is going on here.

The reason they are zero is that crfsuite haven't seen these transitions in training data, and assumed there is no need to learn weights for them, to save some computation time. This is the default behavior, but it is possible to turn it off using sklearn_crfsuite.CRF `all_possible_transitions` option. 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.

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.

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
```

```
Weight  Feature
------  --------------
+4.931  BOS
+3.754  postag[:2]:Fp
+3.539  bias
... 15043 more positive ...
... 3906 more negative ...
-3.685  word.isupper()
-7.025  word.istitle()

y='B-LOC' top features
Weight  Feature
------  ------------------
+2.397  word.istitle()
+2.147  -1:word.lower():en
  ... 2284 more positive ...
  ... 433 more negative ...
-1.080  postag[:2]:SP
-1.080  postag:SP
-1.273  -1:word.istitle()

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

CHAPTER 3

Supported Libraries

scikit-learn

ELI5 supports many estimators, transformers and other components from the scikit-learn library.

Additional `explain_weights` and `explain_prediction` parameters

For all supported scikit-learn classifiers and regressors *`eli5.explain_weights()`* and *`eli5.explain_prediction()`* accept additional keyword arguments. Additional *`eli5.explain_weights()`* parameters:

- `vec` is a vectorizer instance used to transform raw features to the input of the classifier or regressor (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.

Additional *`eli5.explain_prediction()`* parameters:

- `vec` is a vectorizer instance used to transform raw features to the input of the classifier or regressor (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.
- `vectorized` is a flag which tells eli5 if `doc` should be passed through `vec` or not. By default it is False, meaning that if `vec` is not None, `vec.transform([doc])` is passed to the estimator. Set it to False if you're passing `vec` (e.g. to get feature names and/or enable *text highlighting*), but `doc` is already vectorized.

Linear estimators

For linear estimators eli5 maps coefficients back to feature names directly. Supported estimators from sklearn.linear_model:

- ElasticNet
- ElasticNetCV
- HuberRegressor
- Lars

- LarsCV
- Lasso
- LassoCV
- LassoLars
- LassoLarsCV
- LassoLarsIC
- LinearRegression
- LogisticRegression
- LogisticRegressionCV
- OrthogonalMatchingPursuit
- OrthogonalMatchingPursuitCV
- PassiveAggressiveClassifier
- PassiveAggressiveRegressor
- Perceptron
- Ridge
- RidgeClassifier
- RidgeClassifierCV
- RidgeCV
- SGDClassifier
- SGDRegressor
- TheilSenRegressor

Linear SVMs from `sklearn.svm` are also supported:

- LinearSVC
- LinearSVR

For linear scikit-learn classifiers *eli5.explain_weights()* supports one more keyword argument, in addition to common argument and extra arguments for all scikit-learn estimators:

- `coef_scale` is a 1D np.ndarray with a scaling coefficient for each feature; coef[i] = coef[i] * coef_scale[i] if coef_scale[i] is not nan. Use it if you want to scale coefficients before displaying them, to take input feature sign or scale in account.

Note: Top-level *eli5.explain_weights()* and *eli5.explain_prediction()* calls are dispatched to these functions for linear scikit-learn estimators:

- *eli5.sklearn.explain_weights.explain_linear_classifier_weights()*
- *eli5.sklearn.explain_weights.explain_linear_regressor_weights()*
- *eli5.sklearn.explain_prediction.explain_prediction_linear_classifier()*
- *eli5.sklearn.explain_prediction.explain_prediction_linear_regressor()*

Decision Trees, Ensembles

eli5 supports the following tree-based estimators from `sklearn.tree`:

- DecisionTreeClassifier
- DecisionTreeRegressor

eli5.explain_weights() computes feature importances and prepares tree visualization; *eli5.show_weights()* may visualizes a tree either as text or as image (if graphviz is available).

For DecisionTreeClassifier and DecisionTreeRegressor additional *eli5.explain_weights()* keyword arguments are forwarded to sklearn.tree.export_graphviz function when graphviz is available; they can be used to customize tree image.

Note: For decision trees top-level *eli5.explain_weights()* calls are dispatched to *eli5.sklearn.explain_weights.explain_decision_tree()*.

The following tree ensembles from `sklearn.ensemble` are supported:

- GradientBoostingClassifier
- GradientBoostingRegressor
- AdaBoostClassifier (only *eli5.explain_weights()*)
- AdaBoostRegressor (only *eli5.explain_weights()*)
- RandomForestClassifier
- RandomForestRegressor
- ExtraTreesClassifier
- ExtraTreesRegressor

For ensembles *eli5.explain_weights()* computes feature importances and their std deviation.

Note: For ensembles top-level *eli5.explain_weights()* calls are dispatched to *eli5.sklearn.explain_weights.explain_rf_feature_importance()*.

---

eli5.explain_prediction() is less straightforward for ensembles and trees; eli5 uses an approach based on ideas from http://blog.datadive.net/interpreting-random-forests/ : feature weights are calculated by following decision paths in trees of an ensemble (or a single tree for DecisionTreeClassifier and DecisionTreeRegressor). Each node of the tree has an output score, and contribution of a feature on the decision path is how much the score changes from parent to child.

There is a separate package for this explaination method (https://github.com/andosa/treeinterpreter); eli5 implementation is independent.

Note: For decision trees and ensembles *eli5.explain_prediction()* calls are dispatched to *eli5.sklearn.explain_prediction.explain_prediction_tree_classifier()* and *eli5.sklearn.explain_prediction.explain_prediction_tree_regressor()*.

Transformation pipelines

eli5.explain_weights() can be applied to a scikit-learn Pipeline as long as:

- `explain_weights` is supported for the final step of the Pipeline;
- *`eli5.transform_feature_names()`* is supported for all preceding steps of the Pipeline. singledispatch can be used to register `transform_feature_names` for transformer classes not handled (yet) by ELI5 or to override the default implementation.

For instance, imagine a transformer which selects every second feature:

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.utils.validation import check_array
from eli5 import transform_feature_names

class OddTransformer(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        # we store n_features_ for the sake of transform_feature_names
        # when in_names=None:
        self.n_features_ = check_array(X).shape[1]
        return self

    def transform(self, X):
        return check_array(X)[:, 1::2]

@transform_feature_names.register(OddTransformer)
def odd_feature_names(transformer, in_names=None):
    if in_names is None:
        from eli5.sklearn.utils import get_feature_names
        # generate default feature names
        in_names = get_feature_names(transformer, num_features=transformer.n_features_
↪)
    # return a list of strings derived from in_names
    return in_names[1::2]

# Now we can:
#   my_pipeline = make_pipeline(OddTransformer(), MyClassifier())
#   my_pipeline.fit(X, y)
#   explain_weights(my_pipeline)
#   explain_weights(my_pipeline, feature_names=['a', 'b', ...])
```

Note that the `in_names != None` case does not need to be handled as long as the transformer will always be passed the set of feature names either from `explain_weights(my_pipeline, feature_names=...)` or from the previous step in the Pipeline.

Currently the following transformers are supported out of the box:

- any transformer which provides `.get_feature_names()` method;
- nested FeatureUnions and Pipelines;
- SelectorMixin-based transformers: SelectPercentile, SelectKBest, GenericUnivariateSelect, VarianceThreshold, RFE, RFECV, SelectFromModel, RandomizedLogisticRegression.

Reversing hashing trick

eli5 allows to recover feature names for HashingVectorizer and FeatureHasher by computing hashes for the provided example data. *`eli5.explain_prediction()`* handles HashingVectorizer as `vec` automatically; to handle HashingVectorizer and FeatureHasher for *`eli5.explain_weights()`*, use *`InvertableHashingVectorizer`* or *`FeatureUnhasher`*:

```
# vec is a HashingVectorizer instance
# clf is a classifier which works on HashingVectorizer output
# X_sample is a representative sample of input documents

import eli5
from eli5.sklearn import InvertableHashingVectorizer
ivec = InvertableHashingVectorizer(vec)
ivec.fit(X_sample)

# now ``ivec.get_feature_names()`` returns meaningful feature names,
# and ``ivec`` can be used as a vectorizer for eli5.explain_weights:
eli5.explain_weights(clf, vec=ivec)
```

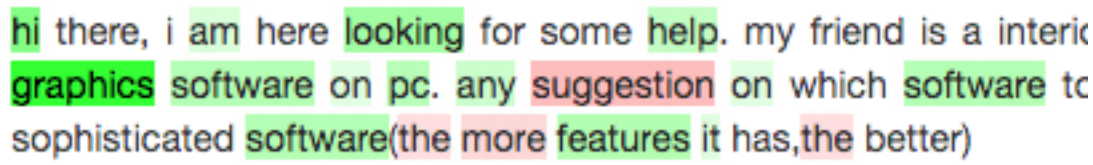
HashingVectorizer is also supported inside a FeatureUnion: *eli5.explain_prediction()* handles this case automatically, and for *eli5.explain_weights()* you can use *eli5.sklearn.unhashing.invert_hashing_and_fit()* (it works for plain HashingVectorizer too) - it tears FeatureUnion apart, inverts and fits all hashing vectorizers and returns a new FeatureUnion:

```
from eli5.sklearn import invert_hashing_and_fit

ivec = invert_hashing_and_fit(vec, X_sample)
eli5.explain_weights(clf, vec=ivec)
```

Text highlighting

For text data *eli5.explain_prediction()* can show the input document with its parts (tokens, characters) highlighted according to their contribution to the prediction result:



It works if the document is vectorized using CountVectorizer, TfIdfVectorizer or HashingVectorizer, and a fitted vectorizer instance is passed to *eli5.explain_prediction()* in a `vec` argument. Custom preprocessors are supported, but custom analyzers or tokenizers are not: highligting works only with 'word', 'char' or 'char_wb' analyzers and a default tokenizer (non-default token_pattern is supported).

Text highlighting also works if a document is vectorized using FeatureUnion with at least one of CountVectorizer, TfIdfVectorizer or HashingVectorizer in the transformer list; features of other transformers are displayed in a regular table.

See also: *Debugging scikit-learn text classification pipeline* tutorial.

OneVsRestClassifier

eli5.explain_weights() and *eli5.explain_prediction()* handle OneVsRestClassifier by dispatching to the explanation function for OvR base estimator, and then calling this function for the OneVsRestClassifier instance. This works in many cases, but not for all. Please report issues to https://github.com/TeamHG-Memex/eli5/issues.

XGBoost

XGBoost is a popular Gradient Boosting library with Python interface. eli5 supports *eli5.explain_weights()* and *eli5.explain_prediction()* for XGBClassifer and XGBRegressor estimators. It is tested for xgboost >= 0.6a2.

eli5.explain_weights() uses feature importances. Additional arguments for XGBClassifer and XGBRegressor:

- `importance_type` is a way to get feature importance. Possible values are:
  - 'gain' - the average gain of the feature when it is used in trees (default)
  - 'weight' - the number of times a feature is used to split the data across all trees
  - 'cover' - the average coverage of the feature when it is used in trees

`target_names` and `target` arguments are ignored.

Note: Top-level *eli5.explain_weights()* calls are dispatched to *eli5.xgboost.explain_weights_xgboost()* for XGBClassifer and XGBRegressor.

For *eli5.explain_prediction()* eli5 uses an approach based on ideas from http://blog.datadive.net/interpreting-random-forests/ : feature weights are calculated by following decision paths in trees of an ensemble. Each node of the tree has an output score, and contribution of a feature on the decision path is how much the score changes from parent to child.

Additional *eli5.explain_prediction()* keyword arguments supported for XGBClassifer and XGBRegressor:

- `vec` is a vectorizer instance used to transform raw features to the input of the estimator `xgb` (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.
- `vectorized` is a flag which tells eli5 if `doc` should be passed through `vec` or not. By default it is False, meaning that if `vec` is not None, `vec.transform([doc])` is passed to the estimator. Set it to False if you're passing `vec`, but `doc` is already vectorized.

See the *tutorial* for a more detailed usage example.

Note: Top-level *eli5.explain_prediction()* calls are dispatched to *eli5.xgboost.explain_prediction_xgboost()* for XGBClassifer and XGBRegressor.

LightGBM

LightGBM is a fast Gradient Boosting framework; it provides a Python interface. eli5 supports *eli5.explain_weights()* and *eli5.explain_prediction()* for `lightgbm.LGBMClassifer` and `lightgbm.LGBMRegressor` estimators. It is tested against LightGBM master branch.

eli5.explain_weights() uses feature importances. Additional arguments for LGBMClassifier and LGBMClassifier:

- `importance_type` is a way to get feature importance. Possible values are:
  - 'gain' - the average gain of the feature when it is used in trees (default)
  - 'split' - the number of times a feature is used to split the data across all trees

  - 'weight' - the same as 'split', for better compatibility with *XGBoost*.

`target_names` and `target` arguments are ignored.

Note: Top-level *`eli5.explain_weights()`* calls are dispatched to *`eli5.lightgbm.explain_weights_lightgbm()`* for `lightgbm.LGBMClassifer` and `lightgbm.LGBMRegressor`.

For *`eli5.explain_prediction()`* eli5 uses an approach based on ideas from http://blog.datadive.net/interpreting-random-forests/ : feature weights are calculated by following decision paths in trees of an ensemble. Each node of the tree has an output score, and contribution of a feature on the decision path is how much the score changes from parent to child.

Additional *`eli5.explain_prediction()`* keyword arguments supported for `lightgbm.LGBMClassifer` and `lightgbm.LGBMRegressor`:

- `vec` is a vectorizer instance used to transform raw features to the input of the estimator `lgb` (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.
- `vectorized` is a flag which tells eli5 if `doc` should be passed through `vec` or not. By default it is False, meaning that if `vec` is not None, `vec.transform([doc])` is passed to the estimator. Set it to False if you're passing `vec`, but `doc` is already vectorized.

Note: Top-level *`eli5.explain_prediction()`* calls are dispatched to `eli5.xgboost.explain_prediction_lightgbm()` for `lightgbm.LGBMClassifer` and `lightgbm.LGBMRegressor`.

---

lightning

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

Using eli5 with estimators from lightning is exactly the same as using it for scikit-learn built-in linear estimators - see *Additional explain_weights and explain_prediction parameters* and *Linear estimators*.

Supported lightning estimators:

- AdaGradClassifier
- AdaGradRegressor
- CDClassifier
- CDRegressor
- FistaClassifier
- FistaRegressor
- LinearSVC
- LinearSVR
- SAGAClassifier
- SAGARegressor
- SAGClassifier
- SAGRegressor

- SDCAClassifier
- SDCARegressor
- SGDClassifier
- SGDRegressor

sklearn-crfsuite

sklearn-crfsuite is a sequence classification library. It provides a higher-level API for python-crfsuite; python-crfsuite is a Python binding for CRFSuite C++ library.

eli5 supports *eli5.explain_weights()* for sklearn_crfsuite.CRF objects; explanation contains transition features table and state features table.

```
import eli5
eli5.explain_weights(crf)
```

See the *tutorial* for a more detailed usage example.

Note: Top-level *eli5.explain_weights()* calls are dispatched to *eli5.sklearn_crfsuite.explain_weights.explain_weights_sklearn_crfsuite()*.

CHAPTER 4

LIME

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.

eli5.lime

To understand how to use `eli5.lime` with text data check the *TextExplainer tutorial*. API reference is available *here*. Currently eli5 doesn't provide a lot of helpers for LIME + non-text data, but there is an IPyhton notebook with an example of applying LIME for such tasks.

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).

   For text data eli5 also provides `eli5.lime.TextExplainer` which brings together all LIME steps and allows to explain text classifiers; it still needs to make assumptions about the classifier in order to generate efficient fake dataset.
3. Similarity metric has a huge effect on a result. By choosing neighbourhood of a different size one can get opposite explanations.

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.

CHAPTER 5

API

API documentation is auto-generated.

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 (more precisely, `re.search(feature_re, x)` is checked).

- **feature_filter** (*Callable[[str], bool], optional*) – Only feature names for which `feature_filter` function returns True 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. Pass a single element, not a one-element array: if you fitted your estimator on `X`, that would be `X[i]` for most containers, and `X.iloc[i]` for `pandas.DataFrame`.

- **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.

- **top_targets** (*int, optional*) – Number of targets to show. When `top_targets` is provided, only specified number of targets with highest scores are shown. Negative value means targets with lowest scores are shown. Must not be given with `targets` argument. `None` value means no limit: all targets are shown (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. Must not be given with `top_targets` argument.

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 (more precisely, `re.search(feature_re, x)` is checked).

- **feature_filter** (*Callable[[str, float], bool], optional*) – Only feature names for which `feature_filter` function returns True are returned. It must accept feature name and feature value. Missing features always have a NaN value.

- ****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 shown (more precisely, `re.search(feature_re, x)` is checked).

- **feature_filter** (*Callable[[str], bool], optional*) – Only feature names for which `feature_filter` function returns True are shown.

- **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.

  `eli5.formatters.fields` provides constants that cover common cases: `INFO` (method and description), `WEIGHTS` (all the rest), and `ALL` (all).

- **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. Pass a single element, not a one-element array: if you fitted your estimator on `X`, that would be `X[i]` for most containers, and `X.iloc[i]` for `pandas.DataFrame`.
- **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.
- **top_targets** (*int, optional*) – Number of targets to show. When `top_targets` is provided, only specified number of targets with highest scores are shown. Negative value means targets with lowest scores are shown. Must not be given with `targets` argument. `None` value means no limit: all targets are shown (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.
- **feature_re** (*str, optional*) – Only feature names which match `feature_re` regex are shown (more precisely, `re.search(feature_re, x)` is checked).
- **feature_filter** (*Callable[[str, float], bool], optional*) – Only feature names for which `feature_filter` function returns True are shown. It must accept feature name and feature value. Missing features always have a NaN value.
- **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.

  `eli5.formatters.fields` provides constants that cover common cases: `INFO` (method and description), `WEIGHTS` (all the rest), and `ALL` (all).

- **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.
- **show_feature_values** (*bool*) – When True, feature values are shown along with feature contributions. Default is False.
- ****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))
```

transform_feature_names(**args*, ***kw*)

Get feature names for transformer output as a function of input names.

Used by *explain_weights()* when applied to a scikit-learn Pipeline, this `singledispatch` should be registered with custom name transformations for each class of transformer.

If there is no `singledispatch` handler registered for a transformer class, `transformer.get_feature_names()` method is called; if there is no such method then feature names are not supported and this function raises an exception.

Parameters

- **transformer** (*scikit-learn-compatible transformer*)
- **in_names** (*list of str, optional*) – Names for features input to transformer.transform(). If not provided, the implementation may generate default feature names if the number of input features is known.

Returns **feature_names** (*list of str*)

eli5.formatters

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*, *show_feature_values=False*)

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. If `show_feature_values` is True, feature values are shown if present. Default is False.

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_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).

render_targets_weighted_spans(*targets*, *preserve_density*)

Return a list of rendered weighted spans for targets. Function must accept a list in order to select consistent weight ranges across all targets.

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.

eli5.formatters.text

format_as_text(*expl*, *show=('method', 'description', 'transition_features', 'targets', 'feature_importances', 'decision_tree')*, *highlight_spaces=None*, *show_feature_values=False*)

Format explanation as text.

Parameters

- **expl** (*eli5.base.Explanation*) – Explanation returned by `eli5.explain_weights` or `eli5.explain_prediction` functions.
- **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.
- **show_feature_values** (*bool*) – When True, feature values are shown along with feature contributions. Default is False.
- **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.

`eli5.formatters.fields` provides constants that cover common cases: `INFO` (method and description), `WEIGHTS` (all the rest), and `ALL` (all).

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.

eli5.lightning

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

eli5.lime

eli5.lime.lime

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

class TextExplainer(*n_samples=5000*, *char_based=None*, *clf=None*, *vec=None*, *sampler=None*, *position_dependent=False*, *rbf_sigma=None*, *random_state=None*, *expand_factor=10*, *token_pattern=None*)
TextExplainer allows to explain predictions of black-box text classifiers using LIME algorithm.

Parameters

- **n_samples** (*int*) – A number of samples to generate and train on. Default is 5000.

  With larger n_samples it takes more CPU time and RAM to explain a prediction, but it could give better results. Larger n_samples could be also required to get good results if you don't want to make strong assumptions about the black-box classifier (e.g. char_based=True and position_dependent=True).

- **char_based** (*bool*) – True if explanation should be char-based, False if it should be token-based. Default is False.

- **clf** (*object, optional*) – White-box probabilistic classifier. It should be supported by eli5, follow scikit-learn interface and provide predict_proba method. When not set, a default classifier is used (logistic regression with elasticnet regularization trained with SGD).
- **vec** (*object, optional*) – Vectorizer which converts generated texts to feature vectors for the white-box classifier. When not set, a default vectorizer is used; which one depends on `char_based` and `position_dependent` arguments.
- **sampler** (*MaskingTextSampler or MaskingTextSamplers, optional*) – Sampler used to generate modified versions of the text.
- **position_dependent** (*bool*) – When True, a special vectorizer is used which takes each token or character (depending on `char_based` value) in account separately. When False (default) a vectorized passed in `vec` or a default vectorizer is used.

  Default vectorizer converts text to vector using bag-of-ngrams or bag-of-char-ngrams approach (depending on `char_based` argument). It means that it may be not powerful enough to approximate a black-box classifier which e.g. takes in account word FOO in the beginning of the document, but not in the end.

  When `position_dependent` is True the model becomes powerful enough to account for that, but it can become more noisy and require larger `n_samples` to get an OK explanation.

  When `char_based=False` the default vectorizer uses word bigrams in addition to unigrams; this is less powerful than `position_dependent=True`, but can give similar results in practice.
- **rbf_sigma** (*float, optional*) – Sigma parameter of RBF kernel used to post-process cosine similarity values. Default is None, meaning no post-processing (cosine simiilarity is used as sample weight as-is). Small `rbf_sigma` values (e.g. 0.1) tell the classifier to pay more attention to generated texts which are close to the original text. Large `rbf_sigma` values (e.g. 1.0) make distance between text irrelevant.

  Note that if you're using large `rbf_sigma` it could be more efficient to use custom `samplers` instead, in order to generate text samples which are closer to the original text in the first place. Use e.g. `max_replace` parameter of *MaskingTextSampler*.
- **random_state** (*integer or numpy.random.RandomState, optional*) – random state
- **expand_factor** (*int or None*) – To approximate output of the probabilistic classifier generated dataset is expanded by `expand_factor` (10 by default) according to the predicted label probabilities. This is a workaround for scikit-learn limitation (no cross-entropy loss for non 1/0 labels). With larger values training takes longer, but probability output can be approximated better.

  expand_factor=None turns this feature off; pass None when you know that black-box classifier returns only 1.0 or 0.0 probabilities.
- **token_pattern** (*str, optional*) – Regex which matches a token. Use it to customize tokenization. Default value depends on `char_based` parameter.

rng_
numpy.random.RandomState – random state

samples_
list[str] – A list of samples the local model is trained on. Only available after *fit()*.

X_
ndarray or scipy.sparse matrix – A matrix with vectorized `samples_`. Only available after *fit()*.

similarity_
ndarray – Similarity vector. Only available after *fit()*.

y_proba_
ndarray – probabilities predicted by black-box classifier (`predict_proba(self.samples_)` result). Only available after *fit()*.

clf_
object – Trained white-box classifier. Only available after *fit()*.

vec_
object – Fit white-box vectorizer. Only available after *fit()*.

metrics_
dict – A dictionary with metrics of how well the local classification pipeline approximates the black-box pipeline. Only available after *fit()*.

explain_prediction(***kwargs*)
Call *eli5.explain_prediction()* for the locally-fit classification pipeline. Keyword arguments are passed to *eli5.explain_prediction()*.

fit() must be called before using this method.

explain_weights(***kwargs*)
Call *eli5.show_weights()* for the locally-fit classification pipeline. Keyword arguments are passed to *eli5.show_weights()*.

fit() must be called before using this method.

fit(*doc*, *predict_proba*)
Explain `predict_proba` probabilistic classification function for the `doc` example. This method fits a local classification pipeline following LIME approach.

To get the explanation use *show_prediction()*, *show_weights()*, *explain_prediction()* or *explain_weights()*.

Parameters

- **doc** (*str*) – Text to explain
- **predict_proba** (*callable*) – Black-box classification pipeline. `predict_proba` should be a function which takes a list of strings (documents) and return a matrix of shape `(n_samples, n_classes)` with probability values - a row per document and a column per output label.

show_prediction(***kwargs*)
Call *eli5.show_prediction()* for the locally-fit classification pipeline. Keyword arguments are passed to *eli5.show_prediction()*.

fit() must be called before using this method.

show_weights(***kwargs*)
Call *eli5.show_weights()* for the locally-fit classification pipeline. Keyword arguments are passed to *eli5.show_weights()*.

fit() must be called before using this method.

eli5.lime.samplers

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

fit (*X=None*, *y=None*)

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

Return (examples, similarity) tuple with generated documents similar to a given document and a vector of similarity values.

class **MaskingTextSampler** (*token_pattern=None*, *bow=True*, *random_state=None*, *replacement=''*, *min_replace=1*, *max_replace=1.0*, *group_size=1*)

Sampler for text data. It randomly removes or replaces tokens from text.

Parameters

- **token_pattern** (*str, optional*) – Regexp for token matching
- **bow** (*bool, optional*) – Sampler could either replace all instances of a given token (bow=True, bag of words sampling) or replace just a single token (bow=False).
- **random_state** (*integer or numpy.random.RandomState, optional*) – random state
- **replacement** (*str*) – Defalt value is '' - by default tokens are removed. If you want to preserve the total token count set `replacement` to a non-empty string, e.g. 'UNKN'.
- **min_replace** (*int or float*) – A minimum number of tokens to replace. Default is 1, meaning 1 token. If this value is float in range [0.0, 1.0], it is used as a ratio. More than min_replace tokens could be replaced if group_size > 1.
- **max_replace** (*int or float*) – A maximum number of tokens to replace. Default is 1.0, meaning all tokens can be replaced. If this value is float in range [0.0, 0.1], it is used as a ratio.
- **group_size** (*int*) – When group_size > 1, groups of nearby tokens are replaced all in once (each token is still replaced with a replacement). Default is 1, meaning individual tokens are replaced.

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

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

class **MaskingTextSamplers** (*sampler_params*, *token_pattern=None*, *random_state=None*, *weights=None*)

Union of MaskingText samplers, with weights. *sample_near()* or *sample_near_with_mask()* generate a requested number of samples using all samplers; a probability of using a sampler is proportional to its weight.

All samplers must use the same token_pattern in order for *sample_near_with_mask()* to work.

Create it with a list of {param: value} dicts with *MaskingTextSampler* paremeters.

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

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

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).

fit(*X*, *y=None*)

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

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.

fit(*X*, *y=None*)

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

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

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*, *random_state=None*, *replacement=''*, *min_replace=1*, *max_replace=1.0*, *group_size=1*)

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).

eli5.sklearn

eli5.sklearn.explain_prediction

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

Explain prediction of a linear classifier.

See *eli5.explain_prediction()* for description of `top`, `top_targets`, `target_names`, `targets`, `feature_names`, `feature_re` and `feature_filter` parameters.

`vec` is a vectorizer instance used to transform raw features to the input of the classifier `clf` (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.

`vectorized` is a flag which tells eli5 if `doc` should be passed through `vec` or not. By default it is False, meaning that if `vec` is not None, `vec.transform([doc])` is passed to the classifier. Set it to False if you're passing `vec`, but `doc` is already vectorized.

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

Explain prediction of a linear regressor.

See *eli5.explain_prediction()* for description of `top`, `top_targets`, `target_names`, `targets`, `feature_names`, `feature_re` and `feature_filter` parameters.

`vec` is a vectorizer instance used to transform raw features to the input of the classifier `clf`; you can pass it instead of `feature_names`.

`vectorized` is a flag which tells eli5 if `doc` should be passed through `vec` or not. By default it is False, meaning that if `vec` is not None, `vec.transform([doc])` is passed to the regressor `reg`. Set it to False if you're passing `vec`, but `doc` is already vectorized.

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

Return an explanation of a scikit-learn estimator

explain_prediction_tree_classifier(*clf*, *doc*, *vec=None*, *top=None*, *top_targets=None*, *target_names=None*, *targets=None*, *feature_names=None*, *feature_re=None*, *feature_filter=None*, *vectorized=False*)

Explain prediction of a tree classifier.

See *eli5.explain_prediction()* for description of `top`, `top_targets`, `target_names`, `targets`, `feature_names`, `feature_re` and `feature_filter` parameters.

`vec` is a vectorizer instance used to transform raw features to the input of the classifier `clf` (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.

`vectorized` is a flag which tells eli5 if `doc` should be passed through `vec` or not. By default it is False, meaning that if `vec` is not None, `vec.transform([doc])` is passed to the classifier. Set it to False if you're passing `vec`, but `doc` is already vectorized.

Method for determining feature importances follows an idea from http://blog.datadive.net/interpreting-random-forests/. Feature weights are calculated by following decision paths in trees of an ensemble (or a single tree for DecisionTreeClassifier). Each node of the tree has an output score, and contribution of a feature on the decision path is how much the score changes from parent to child. Weights of all features sum to the output score or proba of the estimator.

explain_prediction_tree_regressor(*reg*, *doc*, *vec=None*, *top=None*, *top_targets=None*, *target_names=None*, *targets=None*, *feature_names=None*, *feature_re=None*, *feature_filter=None*, *vectorized=False*)

Explain prediction of a tree regressor.

See *eli5.explain_prediction()* for description of `top`, `top_targets`, `target_names`, `targets`, `feature_names`, `feature_re` and `feature_filter` parameters.

`vec` is a vectorizer instance used to transform raw features to the input of the regressor `reg` (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.

`vectorized` is a flag which tells eli5 if `doc` should be passed through `vec` or not. By default it is False, meaning that if `vec` is not None, `vec.transform([doc])` is passed to the regressor. Set it to False if you're passing `vec`, but `doc` is already vectorized.

Method for determining feature importances follows an idea from http://blog.datadive.net/interpreting-random-forests/. Feature weights are calculated by following decision paths in trees of an ensemble (or a single tree for DecisionTreeRegressor). Each node of the tree has an output score, and contribution of a feature on the decision path is how much the score changes from parent to child. Weights of all features sum to the output score of the estimator.

eli5.sklearn.explain_weights

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

Return an explanation of a decision tree.

See *eli5.explain_weights()* for description of `top`, `target_names`, `feature_names`, `feature_re` and `feature_filter` parameters.

`targets` parameter is ignored.

`vec` is a vectorizer instance used to transform raw features to the input of the estimator (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.

All other keyword arguments are passed to sklearn.tree.export_graphviz function.

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

Return an explanation of a linear classifier weights.

See *eli5.explain_weights()* for description of `top`, `target_names`, `targets`, `feature_names`, `feature_re` and `feature_filter` parameters.

`vec` is a vectorizer instance used to transform raw features to the input of the classifier `clf` (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.

`coef_scale` is a 1D np.ndarray with a scaling coefficient for each feature; coef[i] = coef[i] * coef_scale[i] if coef_scale[i] is not nan. Use it if you want to scale coefficients before displaying them, to take input feature sign or scale in account.

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

Return an explanation of a linear regressor weights.

See *eli5.explain_weights()* for description of `top`, `target_names`, `targets`, `feature_names`, `feature_re` and `feature_filter` parameters.

`vec` is a vectorizer instance used to transform raw features to the input of the regressor `reg`; you can pass it instead of `feature_names`.

`coef_scale` is a 1D np.ndarray with a scaling coefficient for each feature; coef[i] = coef[i] * coef_scale[i] if coef_scale[i] is not nan. Use it if you want to scale coefficients before displaying them, to take input feature sign or scale in account.

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

Return an explanation of a tree-based ensemble estimator.

See *eli5.explain_weights()* for description of `top`, `feature_names`, `feature_re` and `feature_filter` parameters.

`target_names` and `targets` parameters are ignored.

`vec` is a vectorizer instance used to transform raw features to the input of the estimator (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.

explain_weights_sklearn(**args*, ***kw*)
Return an explanation of an estimator

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_*.

handle_hashing_vec(*vec*, *feature_names*, *coef_scale*, *with_coef_scale=True*)
Return feature_names and coef_scale (if with_coef_scale is True), calling .get_feature_names for invhashing vectorizers.

invert_hashing_and_fit(*vec*, *docs*)
Create an InvertableHashingVectorizer from hashing vectorizer vec and fit it on docs. If vec is a FeatureUnion, do it for all hashing vectorizers in the union. Return an InvertableHashingVectorizer, or a FeatureUnion, or an unchanged vectorizer.

eli5.sklearn_crfsuite

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

See *eli5.explain_weights()* for description of `top`, `target_names`, `targets`, `feature_re` and `feature_filter` parameters.

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']
```

eli5.xgboost

eli5 has XGBoost support - *eli5.explain_weights()* shows feature importances, and *eli5.explain_prediction()* explains predictions by showing feature weights. Both functions work for XGBClassifier and XGBRegressor.

explain_prediction_xgboost(*xgb*, *doc*, *vec=None*, *top=None*, *top_targets=None*, *target_names=None*, *targets=None*, *feature_names=None*, *feature_re=None*, *feature_filter=None*, *vectorized=False*)
Return an explanation of XGBoost prediction (via scikit-learn wrapper XGBClassifier or XGBRegressor) as feature weights.

See *eli5.explain_prediction()* for description of `top`, `top_targets`, `target_names`, `targets`, `feature_names`, `feature_re` and `feature_filter` parameters.

`vec` is a vectorizer instance used to transform raw features to the input of the estimator `xgb` (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.

`vectorized` is a flag which tells eli5 if `doc` should be passed through `vec` or not. By default it is False, meaning that if `vec` is not None, `vec.transform([doc])` is passed to the estimator. Set it to False if you're passing `vec`, but `doc` is already vectorized.

Method for determining feature importances follows an idea from http://blog.datadive.net/interpreting-random-forests/. Feature weights are calculated by following decision paths in trees of an ensemble. Each leaf has an output score, and expected scores can also be assigned to parent nodes. Contribution of one feature on the decision path is how much expected score changes from parent to child. Weights of all features sum to the output score of the estimator.

explain_weights_xgboost(*xgb*, *vec=None*, *top=20*, *target_names=None*, *targets=None*, *feature_names=None*, *feature_re=None*, *feature_filter=None*, *importance_type='gain'*)

Return an explanation of an XGBoost estimator (via scikit-learn wrapper XGBClassifier or XGBRegressor) as feature importances.

See *eli5.explain_weights()* for description of `top`, `feature_names`, `feature_re` and `feature_filter` parameters.

`target_names` and `targets` parameters are ignored.

Parameters **importance_type** (*str, optional*) – A way to get feature importance. Possible values are:

- 'gain' - the average gain of the feature when it is used in trees (default)
- 'weight' - the number of times a feature is used to split the data across all trees
- 'cover' - the average coverage of the feature when it is used in trees

eli5.lightgbm

eli5 has LightGBM support - *eli5.explain_weights()* shows feature importances, and *eli5.explain_prediction()* explains predictions by showing feature weights. Both functions work for LGBMClassifier and LGBMRegressor.

explain_prediction_lightgbm(*lgb*, *doc*, *vec=None*, *top=None*, *top_targets=None*, *target_names=None*, *targets=None*, *feature_names=None*, *feature_re=None*, *feature_filter=None*, *vectorized=False*)

Return an explanation of LightGBM prediction (via scikit-learn wrapper LGBMClassifier or LGBMRegressor) as feature weights.

See *eli5.explain_prediction()* for description of `top`, `top_targets`, `target_names`, `targets`, `feature_names`, `feature_re` and `feature_filter` parameters.

`vec` is a vectorizer instance used to transform raw features to the input of the estimator `xgb` (e.g. a fitted CountVectorizer instance); you can pass it instead of `feature_names`.

`vectorized` is a flag which tells eli5 if `doc` should be passed through `vec` or not. By default it is False, meaning that if `vec` is not None, `vec.transform([doc])` is passed to the estimator. Set it to False if you're passing `vec`, but `doc` is already vectorized.

Method for determining feature importances follows an idea from http://blog.datadive.net/interpreting-random-forests/. Feature weights are calculated by following decision paths in trees of an ensemble. Each leaf has an output score, and expected scores can also be assigned to parent nodes. Contribution of one feature on the decision path is how much expected score changes from parent to child. Weights of all features sum to the output score of the estimator.

explain_weights_lightgbm(*lgb*, *vec=None*, *top=20*, *target_names=None*, *targets=None*, *feature_names=None*, *feature_re=None*, *feature_filter=None*, *importance_type='gain'*)

Return an explanation of an LightGBM estimator (via scikit-learn wrapper LGBMClassifier or LGBMRegressor) as feature importances.

See *eli5.explain_weights()* for description of `top`, `feature_names`, `feature_re` and `feature_filter` parameters.

`target_names` and `targets` parameters are ignored.

Parameters **importance_type** (*str, optional*) – A way to get feature importance. Possible values are:

- 'gain' - the average gain of the feature when it is used in trees (default)
- 'split' - the number of times a feature is used to split the data across all trees
- 'weight' - the same as 'split', for compatibility with xgboost

eli5.base

class DocWeightedSpans(*document*, *spans*, *preserve_density=None*, *vec_name=None*)

Features highlighted in text. :document: is a pre-processed document before applying the analyzer. :weighted_spans: holds a list of spans for features found in text (span indices correspond to :document:). :preserve_density: determines how features are colored when doing formatting - it is better set to True for char features and to False for word features.

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.

class FeatureImportances(*importances*, *remaining*)

Feature importances with number of remaining non-zero features.

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.

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.

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.

class TransitionFeatureWeights(*class_names*, *coef*)

Weights matrix for transition features.

class TreeInfo(*criterion*, *tree*, *graphviz*, *is_classification*)

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.

class WeightedSpans(*docs_weighted_spans*, *other=None*)

Holds highlighted spans for parts of document - a DocWeightedSpans object for each vectorizer, and other features not highlighted anywhere.

CHAPTER 6

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 7

Changelog

0.6.1 (2017-05-10)

- Better pandas support in *eli5.explain_prediction()* for xgboost, sklearn, LightGBM and lightning.

0.6 (2017-05-03)

- Better scikit-learn Pipeline support in *eli5.explain_weights()*: it is now possible to pass a Pipeline object directly. Curently only SelectorMixin-based transformers, FeatureUnion and transformers with `get_feature_names` are supported, but users can register other transformers; built-in list of supported transformers will be expanded in future. See *Transformation pipelines* for more.
- Inverting of HashingVectorizer is now supported inside FeatureUnion via *eli5.sklearn.unhashing.invert_hashing_and_fit()*. See *Reversing hashing trick*.
- Fixed compatibility with Jupyter Notebook >= 5.0.0.
- Fixed *eli5.explain_weights()* for Lasso regression with a single feature and no intercept.
- Fixed unhashing support in Python 2.x.
- Documentation and testing improvements.

0.5 (2017-04-27)

- LightGBM support: *eli5.explain_prediction()* and *eli5.explain_weights()* are now supported for `LGBMClassifier` and `LGBMRegressor` (see *eli5 LightGBM support*).
- fixed text formatting if all weights are zero;
- type checks now use latest mypy;
- testing setup improvements: Travis CI now uses Ubuntu 14.04.

0.4.2 (2017-03-03)

- bug fix: eli5 should remain importable if xgboost is available, but not installed correctly.

0.4.1 (2017-01-25)

- feature contribution calculation fixed for *eli5.xgboost.explain_prediction_xgboost()*

0.4 (2017-01-20)

- *eli5.explain_prediction()*: new 'top_targets' argument allows to display only predictions with highest or lowest scores;
- *eli5.explain_weights()* allows to customize the way feature importances are computed for XGBClassifier and XGBRegressor using `importance_type` argument (see docs for the *eli5 XGBoost support*);
- *eli5.explain_weights()* uses gain for XGBClassifier and XGBRegressor feature importances by default; this method is a better indication of what's going, and it makes results more compatible with feature importances displayed for scikit-learn gradient boosting methods.

0.3.1 (2017-01-16)

- packaging fix: scikit-learn is added to install_requires in setup.py.

0.3 (2017-01-13)

- *eli5.explain_prediction()* works for XGBClassifier, XGBRegressor from XGBoost and for ExtraTreesClassifier, ExtraTreesRegressor, GradientBoostingClassifier, GradientBoostingRegressor, RandomForestClassifier, RandomForestRegressor, DecisionTreeClassifier and DecisionTreeRegressor from scikit-learn. Explanation method is based on http://blog.datadive.net/interpreting-random-forests/ .
- *eli5.explain_weights()* now supports tree-based regressors from scikit-learn: DecisionTreeRegressor, AdaBoostRegressor, GradientBoostingRegressor, RandomForestRegressor and ExtraTreesRegressor.
- *eli5.explain_weights()* works for XGBRegressor;
- new *TextExplainer* class allows to explain predictions of black-box text classification pipelines using LIME algorithm; many improvements in *eli5.lime*.
- better `sklearn.pipeline.FeatureUnion` support in *eli5.explain_prediction()*;
- rendering performance is improved;
- a number of remaining feature importances is shown when the feature importance table is truncated;
- styling of feature importances tables is fixed;
- *eli5.explain_weights()* and *eli5.explain_prediction()* support more linear estimators from scikit-learn: HuberRegressor, LarsCV, LassoCV, LassoLars, LassoLarsCV, LassoLarsIC, OrthogonalMatchingPursuit, OrthogonalMatchingPursuitCV, PassiveAggressiveRegressor, RidgeClassifier, RidgeClassifierCV, TheilSenRegressor.

- text-based formatting of decision trees is changed: for binary classification trees only a probability of "true" class is printed, not both probabilities as it was before.
- *eli5.explain_weights()* supports `feature_filter` in addition to `feature_re` for filtering features, and *eli5.explain_prediction()* now also supports both of these arguments;
- 'Weight' column is renamed to 'Contribution' in the output of *eli5.explain_prediction()*;
- new `show_feature_values=True` formatter argument allows to display input feature values;
- fixed an issue with analyzer='char_wb' highlighting at the start of the text.

0.2 (2016-12-03)

- XGBClassifier support (from XGBoost package);
- *eli5.explain_weights()* support for sklearn OneVsRestClassifier;
- std deviation of feature importances is no longer printed as zero if it is not available.

0.1.1 (2016-11-25)

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

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.

0.0.6 (2016-10-12)

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

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.

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.

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.

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.

0.0.1 (2016-09-15)

Pre-release.

License is MIT.

Python Module Index

e

eli5, 43
eli5.base, 60
eli5.formatters, 48
eli5.formatters.as_dict, 50
eli5.formatters.html, 49
eli5.formatters.text, 49
eli5.lightgbm, 59
eli5.lightning, 50
eli5.lime, 50
eli5.lime.lime, 50
eli5.lime.samplers, 52
eli5.lime.textutils, 54
eli5.sklearn.explain_prediction, 54
eli5.sklearn.explain_weights, 56
eli5.sklearn.unhashing, 57
eli5.sklearn_crfsuite.explain_weights, 58
eli5.xgboost, 58

Index

B

BaseSampler (class in eli5.lime.samplers), 52

C

clf_ (TextExplainer attribute), 52
column_signs_ (InvertableHashingVectorizer attribute), 57
cosine_similarity_vec() (in module eli5.lime.textutils), 54

D

DocWeightedSpans (class in eli5.base), 60

E

eli5 (module), 43
eli5.base (module), 60
eli5.formatters (module), 48
eli5.formatters.as_dict (module), 50
eli5.formatters.html (module), 49
eli5.formatters.text (module), 49
eli5.lightgbm (module), 59
eli5.lightning (module), 50
eli5.lime (module), 50
eli5.lime.lime (module), 50
eli5.lime.samplers (module), 52
eli5.lime.textutils (module), 54
eli5.sklearn.explain_prediction (module), 54
eli5.sklearn.explain_weights (module), 56
eli5.sklearn.unhashing (module), 57
eli5.sklearn_crfsuite.explain_weights (module), 58
eli5.xgboost (module), 58
explain_decision_tree() (in module eli5.sklearn.explain_weights), 56
explain_linear_classifier_weights() (in module eli5.sklearn.explain_weights), 56
explain_linear_regressor_weights() (in module eli5.sklearn.explain_weights), 56
explain_prediction() (in module eli5), 44
explain_prediction() (TextExplainer method), 52
explain_prediction_lightgbm() (in module eli5.lightgbm), 59
explain_prediction_lightning() (in module eli5.lightning), 50
explain_prediction_linear_classifier() (in module eli5.sklearn.explain_prediction), 54
explain_prediction_linear_regressor() (in module eli5.sklearn.explain_prediction), 55
explain_prediction_sklearn() (in module eli5.sklearn.explain_prediction), 55
explain_prediction_tree_classifier() (in module eli5.sklearn.explain_prediction), 55
explain_prediction_tree_regressor() (in module eli5.sklearn.explain_prediction), 55
explain_prediction_xgboost() (in module eli5.xgboost), 58
explain_rf_feature_importance() (in module eli5.sklearn.explain_weights), 56
explain_weights() (in module eli5), 43
explain_weights() (TextExplainer method), 52
explain_weights_lightgbm() (in module eli5.lightgbm), 60
explain_weights_lightning() (in module eli5.lightning), 50
explain_weights_sklearn() (in module eli5.sklearn.explain_weights), 57
explain_weights_sklearn_crfsuite() (in module eli5.sklearn_crfsuite.explain_weights), 58
explain_weights_xgboost() (in module eli5.xgboost), 59
Explanation (class in eli5.base), 60

F

FeatureImportances (class in eli5.base), 60
FeatureUnhasher (class in eli5.sklearn.unhashing), 57
FeatureWeights (class in eli5.base), 60
filter_transition_coefs() (in module eli5.sklearn_crfsuite.explain_weights), 58
fit() (BaseSampler method), 52
fit() (InvertableHashingVectorizer method), 57
fit() (MultivariateKernelDensitySampler method), 54

fit() (TextExplainer method), 52
fit() (UnivariateKernelDensitySampler method), 54
format_as_dict() (in module eli5.formatters.as_dict), 50
format_as_html() (in module eli5.formatters.html), 49
format_as_text() (in module eli5.formatters.text), 49
format_hsl() (in module eli5.formatters.html), 49
format_html_styles() (in module eli5.formatters.html), 49

G

generate_samples() (in module eli5.lime.textutils), 54
get_feature_names() (InvertableHashingVectorizer method), 57
get_weight_range() (in module eli5.formatters.html), 49

H

handle_hashing_vec() (in module eli5.sklearn.unhashing), 57

I

invert_hashing_and_fit() (in module eli5.sklearn.unhashing), 58
InvertableHashingVectorizer (class in eli5.sklearn.unhashing), 57

M

MaskingTextSampler (class in eli5.lime.samplers), 53
MaskingTextSamplers (class in eli5.lime.samplers), 53
metrics_ (TextExplainer attribute), 52
MultivariateKernelDensitySampler (class in eli5.lime.samplers), 53

N

NodeInfo (class in eli5.base), 60

R

recalculate_attributes() (FeatureUnhasher method), 57
remaining_weight_color_hsl() (in module eli5.formatters.html), 49
render_targets_weighted_spans() (in module eli5.formatters.html), 49
rng_ (TextExplainer attribute), 51

S

sample_near() (BaseSampler method), 53
sample_near() (MaskingTextSampler method), 53
sample_near() (MaskingTextSamplers method), 53
sample_near() (MultivariateKernelDensitySampler method), 54
sample_near() (UnivariateKernelDensitySampler method), 54
sample_near_with_mask() (MaskingTextSampler method), 53
sample_near_with_mask() (MaskingTextSamplers method), 53
samples_ (TextExplainer attribute), 51
show_prediction() (in module eli5), 46
show_prediction() (TextExplainer method), 52
show_weights() (in module eli5), 45
show_weights() (TextExplainer method), 52
similarity_ (TextExplainer attribute), 51
sorted_for_ner() (in module eli5.sklearn_crfsuite.explain_weights), 58

T

TargetExplanation (class in eli5.base), 60
TextExplainer (class in eli5.lime.lime), 50
transform_feature_names() (in module eli5), 48
TransitionFeatureWeights (class in eli5.base), 60
TreeInfo (class in eli5.base), 60

U

UnivariateKernelDensitySampler (class in eli5.lime.samplers), 54

V

vec_ (TextExplainer attribute), 52

W

weight_color_hsl() (in module eli5.formatters.html), 49
WeightedSpans (class in eli5.base), 60

X

X_ (TextExplainer attribute), 51

Y

y_proba_ (TextExplainer attribute), 52