Diving into data

A blog on machine learning, data mining and visualization

About

Random forest interpretation with scikit-learn

POSTED AUGUST 12, 2015

In one of my previous posts I discussed how random forests can be turned into a "white box", such that each prediction is decomposed into a sum of contributions from each feature i.e. $prediction = bias + feature_1 contribution + \ldots + feature_n contribution$.

I've a had quite a few requests for code to do this. Unfortunately, most random forest libraries (including scikit-learn) don't expose tree paths of predictions. The implementation for sklearn required a hacky patch for exposing the paths. Fortunately, since 0.17.dev, scikit-learn has two additions in the API that make this relatively straightforward: obtaining leaf node_ids for predictions, and storing all intermediate values in all nodes in decision trees, not only leaf nodes. Combining these, it is possible to extract the prediction paths for each individual prediction and decompose the predictions via inspecting the paths.

Without further ado, the code is available at github, and also via `pip install treeinterpreter`

Note: this requires scikit-learn 0.17, which is still in development. You can check how to install it at http://scikit-learn.org/stable/install.html#install-bleeding-edge

Decomposing random forest predictions with treeinterpreter

Let's take a sample dataset, train a random forest model, predict some values on the test set and then decompose the predictions.

```
from treeinterpreter import treeinterpreter as ti
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
import numpy as np

from sklearn.datasets import load_boston
boston = load_boston()
rf = RandomForestRegressor()
rf.fit(boston.data[:300], boston.target[:300])
```

Lets pick two arbitrary data points that yield different price estimates from the model.

```
instances = boston.data[[300, 309]]
print "Instance 0 prediction:", rf.predict(instances[0])
print "Instance 1 prediction:", rf.predict(instances[1])
```

```
 Instance 0 prediction: [ 30.76]
 Instance 1 prediction: [ 22.41]
```

Predictions that the random forest model made for the two data points are quite different. But why? We can now decompose the predictions into the bias term (which is just the trainset mean) and individual feature contributions, so we see which features contributed to the difference and by how much.

We can simply call the treeinterpreter `predict` method with the model and the data.

```
prediction, bias, contributions = ti.predict(rf, instances)
```

Printint out the results:

```
for i in range(len(instances)):
    print "Instance", i
    print "Bias (trainset mean)", biases[i]
    print "Feature contributions:"
    for c, feature in sorted(zip(contributions[i],
                                 boston.feature_names),
                             key=lambda x: -abs(x[0])):
        print feature, round(c, 2)
    print "-"*20
```

```
Instance 0
Bias (trainset mean) 25.2849333333
Feature contributions:
RM 2.73
LSTAT 1.71
PTRATIO 1.27
ZN 1.04
DIS -0.7
B -0.39
TAX -0.19
CRIM -0.13
RAD 0.11
INDUS 0.06
AGE -0.02
NOX -0.01
CHAS 0.0
--------------------
Instance 1
Bias (trainset mean) 25.2849333333
Feature contributions:
RM -4.88
LSTAT 2.38
DIS 0.32
AGE -0.28
TAX -0.23
CRIM 0.16
PTRATIO 0.15
B -0.15
INDUS -0.14
CHAS -0.1
ZN -0.05
NOX -0.05
RAD -0.02
```

The feature contributions are sorted by their absolute impact. We can see that in the first instance where the prediction was high, most of the positive contributions came from RM, LSTAT and PTRATIO feaures. On the second instance the predicted value is much lower, since RM feature actually has a very large negative impact that is not offset by the positive impact of other features, thus taking the prediction below the dataset mean.

But is the decomposition actually correct? This is easy to check: bias and contributions need to sum up to the predicted value:

```
print prediction
print biases + np.sum(contributions, axis=1)
```

```
[ 30.76 22.41]
[ 30.76 22.41]
```

Note that when summing up the contributions, we are dealing with floating point numbers so the values can slightly different due to rounding errors

Comparing too datasets

One use case where this approach can be very useful is when comparing two datasets. For example

- Understanding the exact reasons why estimated values are different on two datasets, for example what contributes to estimated house prices being different in two neighborhoods.
- Debugging models and/or data, for example understanding why average predicted values on newer data do not match the results seen on older data.

For this example, let’s split the remaining housing price data into two test datasets and compute the average estimated prices for them.

```
ds1 = boston.data[300:400]
ds2 = boston.data[400:]

print np.mean(rf.predict(ds1))
print np.mean(rf.predict(ds2))
```

```
22.1912
18.4773584906
```

We can see that the average predicted prices for the houses in the two datasets are quite different. We can now trivially break down the contributors to this difference: which features contribute to this different and by how much.

```
prediction1, bias1, contributions1 = ti.predict(rf, ds1)
prediction2, bias2, contributions2 = ti.predict(rf, ds2)
```

We can now calculate the mean contribution of each feature to the difference.

```
totalc1 = np.mean(contributions1, axis=0)
totalc2 = np.mean(contributions2, axis=0)
```

Since biases are equal for both datasets (because the training data for the model was the same), the difference between the average predicted values has to come only from feature contributions. In other words, the sum of the feature contribution differences should be equal to the difference in average prediction, which we can trivially check to be the case

```
print np.sum(totalc1 - totalc2)
print np.mean(prediction1) - np.mean(prediction2)
```

```
3.71384150943
3.71384150943
```

Finally, we can just print out the differences of the contributions in the two datasets. The sum of these is exactly the difference between the average predictions.

```
for c, feature in sorted(zip(totalc1 - totalc2,
                             boston.feature_names), reverse=True):
    print feature, round(c, 2)
```

```
LSTAT 2.8
CRIM 0.5
RM 0.5
PTRATIO 0.09
AGE 0.08
NOX 0.03
B 0.01
CHAS -0.01
ZN -0.02
RAD -0.03
INDUS -0.03
TAX -0.08
DIS -0.14
```

Classification trees and forests

Exactly the same method can be used for classification trees, where features contribute to the estimated probability of a given class.
We can see this on the sample iris dataset.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
iris = load_iris()

rf = RandomForestClassifier(max_depth = 4)
idx = range(len(iris.target))
np.random.shuffle(idx)

rf.fit(iris.data[idx][:100], iris.target[idx][:100])
```

Let’s predict now for a single instance.

```
instance = iris.data[idx][100:101]
print rf.predict_proba(instance)
```

Breakdown of feature contributions:

```
prediction, bias, contributions = ti.predict(rf, instance)
print "Prediction", prediction
print "Bias (trainset prior)", bias
print "Feature contributions:"
for c, feature in zip(contributions[0],
                             iris.feature_names):
    print feature, c
```

```
Prediction [[ 0. 0.9 0.1]]
Bias (trainset prior) [[ 0.36 0.262 0.378]]
Feature contributions:
sepal length (cm) [-0.1228614 0.07971035 0.04315104]
sepal width (cm) [ 0. -0.01352012 0.01352012]
petal length (cm) [-0.11716058 0.24709886 -0.12993828]
petal width (cm) [-0.11997802 0.32471091 -0.20473289]
```

We can see that the strongest contributors to predicting the second class were petal length and width, which had the larges impact on updating the prior.

Summary

Making random forest predictions interpretable is actually pretty straightforward, and leading to similar level of interpretability as linear models. With treeinterpreter (`pip install treeinterpreter`), this can be done with just a couple of lines of code.

Follow @crossentropy
Tweet

28 COMMENTS ON “RANDOM FOREST INTERPRETATION WITH SCIKIT-LEARN”

Pedro Fonseca on **August 13, 2015 at 10:37 am** said:

You, sir, are awesome.

Reply ↓

Pingback: Python learning resources | Wei Shen's Note

Pingback: Distilled News | Data Analytics & R

Pingback: 10 tutorials về scikit-learn và pandas | Ông Xuân Hồng

Pingback: 10 tutorials về scikit-learn | Ông Xuân Hồng

Ravi on **December 14, 2015 at 11:23 pm** said:

Is there a paper to cite if I use this method and library to interpret a random forest?

Reply ↓

ando
on **December 16, 2015 at 8:58 am** said:

Unfortunately there is no paper. You can cite the blog post (a la http://content.easybib.com/citation-guides/mla-format/how-to-cite-a-blog-mla/)

Reply ↓

Ravi
on **January 12, 2016 at 1:51 pm** said:

Thank you.

Reply ↓

Ilya on **December 20, 2015 at 7:52 pm** said:

A small typo: in the section “Decomposing random forest predictions with treeinterpreter” in the third codeblock it should be
1: prediction, biases, contributions = ti.predict(rf, instances)
instead of
1: prediction, bias, contributions = ti.predict(rf, instances)

Reply ↓

Pingback: 用scikit-learn 来演绎随机森林方法 - IT大道

Pingback: 用scikit-learn 来演绎随机森林方法 - IT大道

Anonymous on **April 10, 2016 at 10:47 am** said:

excellent article!

Reply ↓

Marco Tulio Ribeiro on April 14, 2016 at 9:24 pm said:

This is an interesting technique. I wonder if the level of interpretability here can be compared to that of linear models, though. In a linear model, the contribution is completely faithful to the model – i.e. by looking at the weights, one can understand what would change exactly if the feature had a different value.

Since you are explaining trees (which may be completely non-linear), the contribution of a feature may not correspond to the effect it has in the actual model. The XOR case is a good example of this. Imagine the following tree, with values in parenthesis, and where 'right' indicates 'true':

X1? (value = 0.5)
/ \
X2? (value=0.5) X2? (value = 0.5)
/ \ / \
value = 1 value = 0 value = 0 value = 1

For any prediction made by this tree, your method will indicate that X1 has importance 0, and X2 and the bias term each have importance 0.5, which would give the false impression that X1 is not important in the prediction. If we had X2 in the root instead of X1, the importance would completely flip, even though the tree would be equivalent.

In light of this, I wonder what the interpretation of these importance terms really should be – i.e. what do they actually convey?

Still, very interesting :)

Reply ↓

Marco Tulio Ribeiro
on April 14, 2016 at 9:25 pm said:

Sorry, posting removed my formatting and made the tree hard to read. I hope it still makes sense.

Reply ↓

ando
on April 15, 2016 at 1:13 pm said:

You said linear regressor is better in terms of contributors being faithful to the model, how would that faithfulness fare in the XOR example? 😉

Joking aside, you are of course right that with complex feature interactions, taking the sum of individual contributions inherently cannot express such relationships. That's a tradeoff though. The path interpretation method isn't actually bound to using individual features. You can for example compute contributions from feature interactions (condition X followed by condition Y contributes by amount Z), from conditions instead of features (X > 3 contributes amount Z) etc.
So in the end it boils down to your data and the model and what sort of tradeoff you want to have in terms of the complexity and expressiveness of your contribution list.

Reply ↓

Student on **July 10, 2016 at 5:32 pm** said:

Thanks for great article.
I am wondering if you could let me know is it possible to install treeinterpreter by conda?

Reply ↓

Roel on **July 13, 2016 at 4:50 pm** said:

Hi Ando,

Thanks for this useful package. I was hoping to try it out on Gradient Boosting Trees, but I see there is a bug in using the package on it. It has to do with the variable n_outputs, see the error message below.

Do you thing this is something we can fix?

Thanks,
Roel

```
———————————————————————————
AttributeError Traceback (most recent call last)
in ()
17 print "Instance 1 prediction:", gbt.predict(instances[1])
18
—> 19 prediction, bias, contributions = ti.predict(gbt, instances)

/Users/…/anaconda/lib/python2.7/site-packages/treeinterpreter/treeinterpreter.pyc in predict(model, X)
121 """
122 # Only single out response variable supported,
–> 123 if model.n_outputs_ > 1:
124 raise ValueError("Multilabel classification trees not supported")
125

AttributeError: 'GradientBoostingRegressor' object has no attribute 'n_outputs_'
```

Reply ↓

Jay on **September 29, 2016 at 12:04 am** said:

Enjoyed the article. Two questions I can't quite figure out.

1. Using a Random Forest Classifier, the contributions term gives an array of feature contributions by class for each input instance. However, I'm not certain which column refers to which class. Could you clarify?

2. Is it possible to take the mean of all input instances for each feature by class? In other words, which features contribute most to class determination as a whole rather than by instance. Thanks.

Reply ↓

Lao on **October 1, 2016 at 1:18 am** said:

sklearn 0.18 release add decision_path method to RandomForestRegressor, that’s a good news！

Reply ↓

Pingback: Подборка ссылок для изучения Python — IT-News.club

Wei on **November 16, 2016 at 5:24 pm** said:

Dear author,

Is it possible to inspect the class distribution in every internal and leaf node along the decision path?

Reply ↓

Shailendra on **November 18, 2016 at 12:05 pm** said:

How do we accomplish this in R?

Reply ↓

Paul
on **April 4, 2017 at 8:23 am** said:

I really like that approach, in general I think there is gap of interpretation methods of predictions, a pity for me that the implementation is not available in R.

Reply ↓

Pingback: Hot reads for this week in machine learning and deep learning – Everything Artificial Intelligence

Pingback: Подборка ресурсов для изучения Python — IT-News.club

Ido on **January 4, 2017 at 9:30 am** said:

Thanks a lot for this. Do you think this can be done as well for tree-based gradient boosting methods, such as XGBOOST?

Reply ↓

Franco on **November 8, 2017 at 9:02 am** said:

Turning Random Forest from a black-box to a white-box (maybe gray?) is pure GENIUS !!!!!

Reply ↓

Vik on **November 22, 2017 at 9:32 pm** said:

Hi,

I am using H2O Python module(Random Forest). Do you know if I can use treeinterpreter with that?

if no, can you suggest how I might be able to integrate it with H20 Python random forest predictions?

Reply ↓

Leave a Reply

Your email address will not be published.

Comment

Name

Email

Website

Post Comment