W4995 Applied Machine Learning

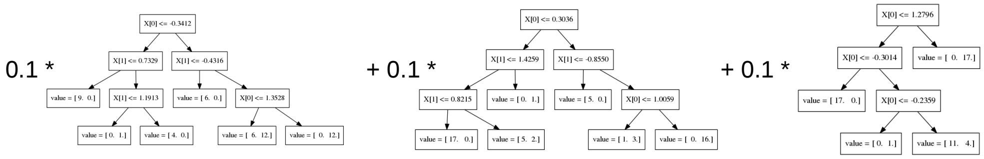
Boosting, Stacking, Calibration

02/22/17

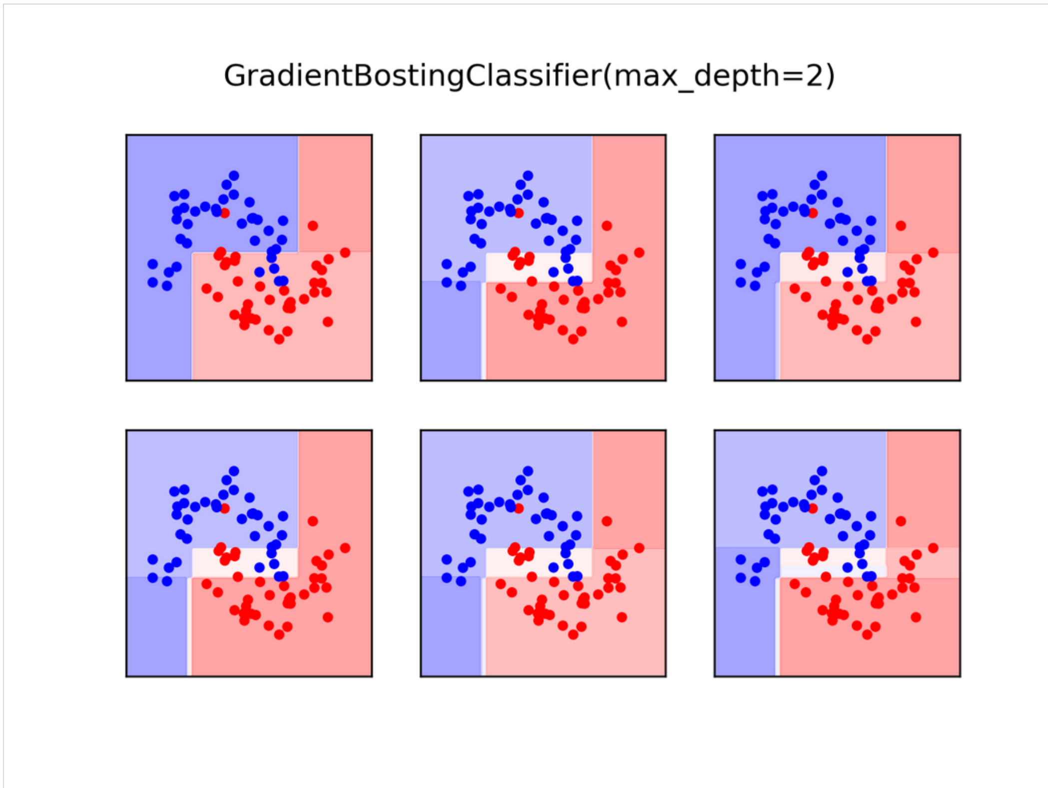
Andreas Müller

Gradient Boosting

Gradient Boosting Algorithm



- Iteratively add regression trees to model
- Use log loss for classification
- Discount update by learning rate


GradientBostingClassifier(max_depth=2)

Gradient Boosting

- Many shallow trees
- learning_rate ↔ n_estimators
- Slower to train than RF (serial), but much faster to predict
- Small model size
- Uses one-vs-rest for multi-class!

Tuning Gradient Boosting

- Pick n_estimators, tune learning rate
- Can also tune max_features
- Typically strong pruning via max_depth

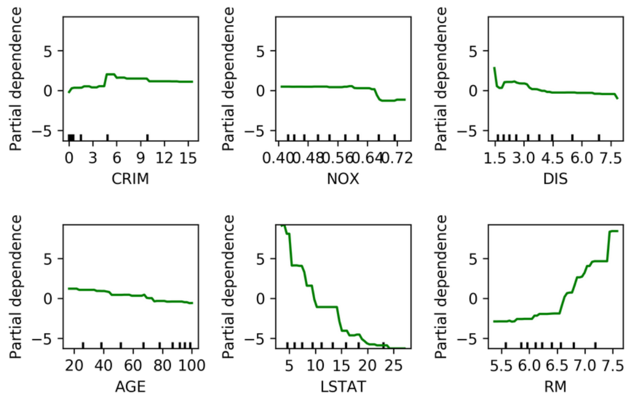
Partial Dependence Plots

- Marginal dependence of prediction on one or two features

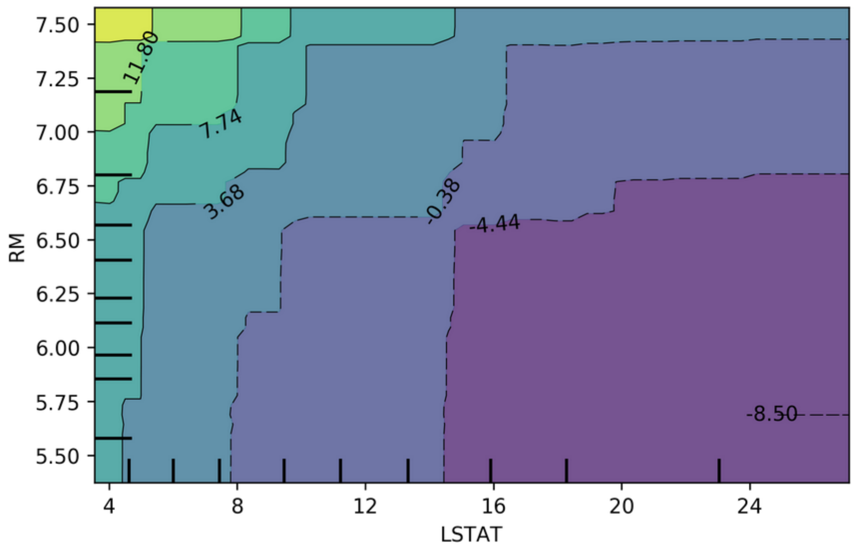
```
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(
    boston.data, boston.target, random_state=0)

gbrt = GradientBoostingRegressor().fit(X_train, y_train)
gbrt.score(X_test, y_test)
```

```
0.81962024379538989
```

```
from sklearn.ensemble.partial_dependence import plot_partial_dependence
fig, axs = plot_partial_dependence(gbrt, X_train, np.argsort(gbrt.feature_importances_)[-6:],
                                   feature_names=boston.feature_names,
                                   n_jobs=3, grid_resolution=50)
plt.tight_layout()
```

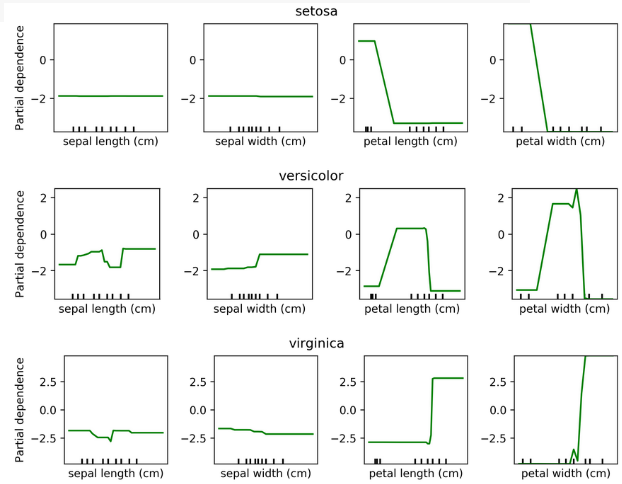


```
fig, axs = plot_partial_dependence(gbrt, X_train, [np.argsort(gbrt.feature_importances_)[-2:]],
                                   feature_names=boston.feature_names,
                                   n_jobs=3, grid_resolution=50)
```



Partial Dependence for Classification

```
from sklearn.ensemble.partial_dependence import plot_partial_dependence
for i in range(3):
    fig, axs = plot_partial_dependence(gbrt, X_train, range(4), n_cols=4,
                                       feature_names=iris.feature_names, grid_resolution=50, label=i,
                                       figsize=(8, 2))
    fig.suptitle(iris.target_names[i])
    for ax in axs: ax.set_xticks(())

    plt.tight_layout()
```



XGBoost

- Efficient implementation of gradient boosting
- Improvements on original algorithm
- https://arxiv.org/abs/1603.02754
- Adds l1 and l2 penalty on leaf-weights
- Fast approximate split finding
- Can pip-install
- Scikit-learn compatible interface

Boosting in General

- “Meta-algorithm” to create strong learners from weak learners.
- AdaBoost, GentleBoost, ...
- Trees or stumps work best
- Gradient Boosting often the best of the bunch
- Many specialized algorithms (ranking etc)

When to use tree-based models

- Model non-linear relationships
- Single tree: very interpretable (if small)
- Random forests very robust, good benchmark
- Gradient boosting often best performance with careful tuning
- Doesn't care about scaling, no need for feature engineering!

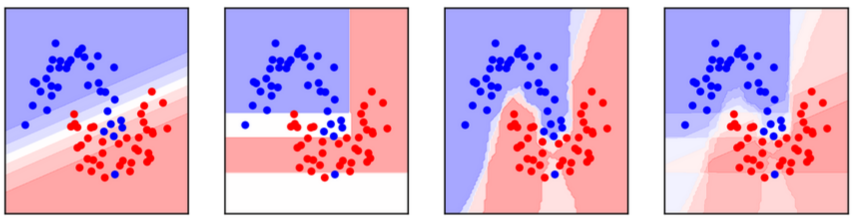
More ensembles: Stacking

Poor man's Stacking

- Build multiple models
- Train model on probabilities / scores produced

```
from sklearn.neighbors import KNeighborsClassifier

X, y = make_moons(noise=.2, random_state=18)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=0)

voting = VotingClassifier([('logreg', LogisticRegression(C=100)),
                           ('tree', DecisionTreeClassifier(max_depth=3, random_state=0)),
                           ('knn', KNeighborsClassifier(n_neighbors=3))
                          ],
                         voting='soft')
voting.fit(X_train, y_train)
lr, tree, knn = voting.estimators_
print(("{:.2f} " * 4).format(voting.score(X_test, y_test),
                             lr.score(X_test, y_test), tree.score(X_test, y_test),
                             knn.score(X_test, y_test)))
```

```
0.88 0.84 0.80 1.00
```



Poor man's Stacking

```
from sklearn.preprocessing import FunctionTransformer
# we need to reshape the result from votingclassifier.transform because
# of some annoyance in sklearn. We then keep only the probabilities of the positive classes!
reshaper = FunctionTransformer(lambda X_: np.rollaxis(X_, 1).reshape(-1, 6)[:, 1::2], validate=False)
stacking = make_pipeline(voting, reshaper,
                         LogisticRegression(C=100))
stacking.fit(X_train, y_train)
stacking.score(X_train, y_train)
```

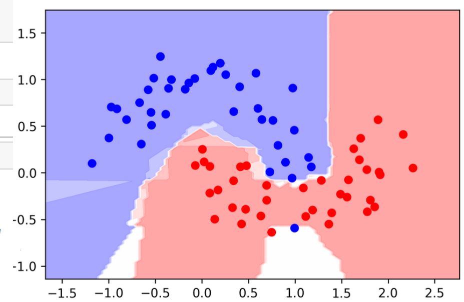
```
0.98666666666666669
```

```
stacking.score(X_test, y_test)
```

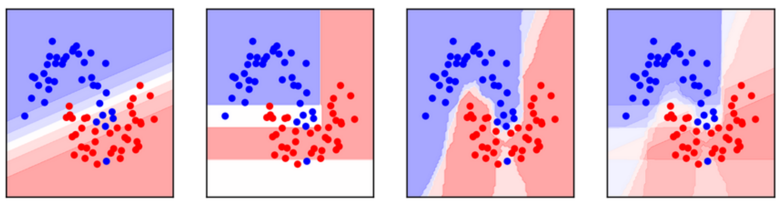
```
0.95999999999999996
```

```
stacking.named_steps['logisticregression'].coef_
```

```
array([[-2.625,  6.261,  9.501]])
```



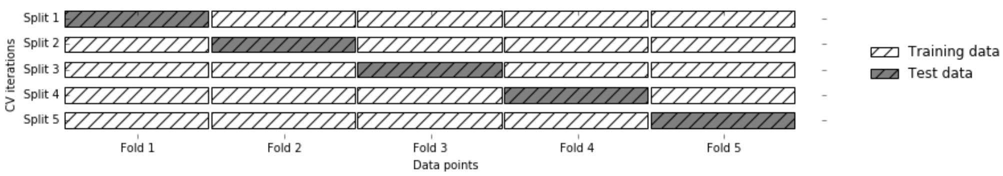
Problem: Overfitting!



Stacking

- Use cross-validation (even LOO!) to produce probability estimates on training set.
- Train second step estimator on held-out estimates
- No overfitting of second step!
- For testing: as usual

Hold-out estimates of probabilities



- Split 1 produces probabilities for Fold 1, split2 for Fold 2 etc.
- Get a probability estimate for each data point!
- Unbiased estimates (like on the test set) for the whole training set!
- Without it: The best estimator is the one that memorized the training set.

Stacking with Scikit-learn

```
from sklearn.model_selection import cross_val_predict
first_stage = make_pipeline(voting, reshaper)
transform_cv = cross_val_predict(first_stage, X_train, y_train, cv=10, method="transform")
```

```
second_stage = LogisticRegression(C=100).fit(transform_cv, y_train)
print(second_stage.coef_)
```

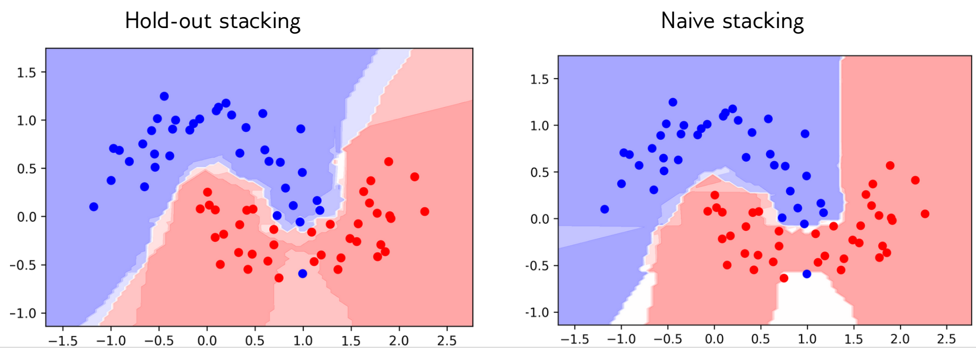
```
[[ 2.09  -1.424  7.93 ]]
```

```
second_stage.score(transform_cv, y_train)
```

```
0.95999999999999996
```

```
second_stage.score(first_stage.transform(X_test), y_test)
```

```
1.0
```



Calibration

http://www.datascienceassn.org/sites/default/files/Predicting%20good%20probabilities%20with%20supervised%20learning.pdf

Probabilities can be much more informative than labels:

"The model predicted you don't have cancer"

vs

"The model predicted you're 40% likely to have cancer"

What and Why of calibration

- Determining **reliable** class distributions for classification.
- Important for decision making!
- "Model has predict_proba" != "good probabilities"

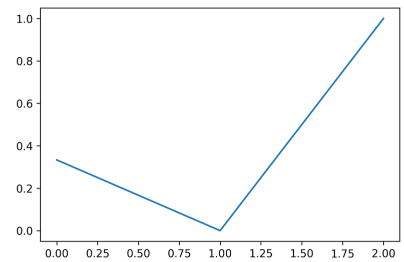
Calibration curve (Reliability diagram)

- For binary classification only
- Given a predicted ranking or probability from a supervised classifier, bin predictions.
- Plot fraction of data that's positive in each bin.
- Doesn't require ground truth probabilities!

$\hat{p}(y)$	y
0.9	1
0.4	0
0.3	1
0.6	0
0.8	1
0.2	0
0.3	0

sort $\rightarrow$

$\hat{p}(y)$	y	bin
0.9	1	1
0.8	1	
0.6	0	0
0.4	0	
0.3	1	1/3
0.3	0	
0.2	0	



calibration_curve with sklearn

Using a subsample of the covertype dataset

```
from sklearn.linear_model import LogisticRegressionCV
print(X_train.shape)
print(np.bincount(y_train))
lr = LogisticRegressionCV().fit(X_train, y_train)
```

```
(52292, 54)
[19036 33256]
```

```
lr.C_
```

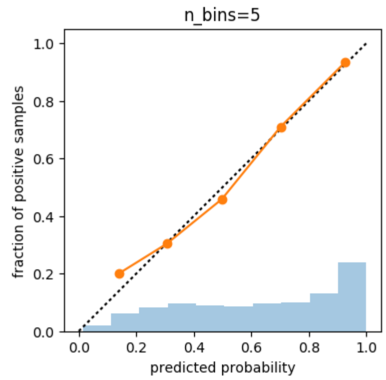
```
array([ 2.783])
```

```
print(lr.predict_proba(X_test)[:10])
print(y_test[:10])
```

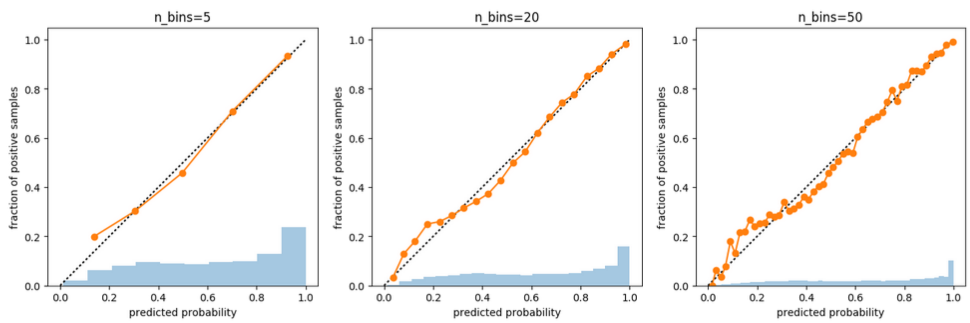
```
[[ 0.681  0.319]
 [ 0.049  0.951]
 [ 0.706  0.294]
 [ 0.537  0.463]
 [ 0.819  0.181]
 [ 0.     1.   ]
 [ 0.794  0.206]
 [ 0.676  0.324]
 [ 0.727  0.273]
 [ 0.597  0.403]]
[0 1 0 1 1 1 0 0 0 1]
```

```
from sklearn.calibration import calibration_curve
probs = lr.predict_proba(X_test)[:, 1]
prob_true, prob_pred = calibration_curve(y_test, probs, n_bins=5)
print(prob_true)
print(prob_pred)
```

```
[ 0.2    0.303  0.458  0.709  0.934]
[ 0.138  0.306  0.498  0.701  0.926]
```

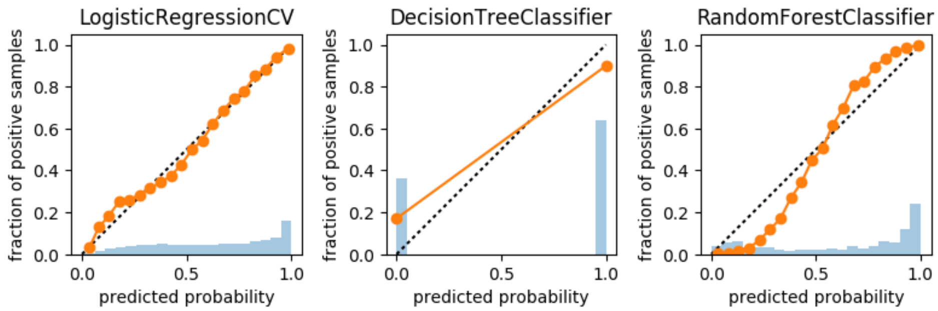


Influence of number of bins



Works here because dataset is big
might become very noisy for larger datasets

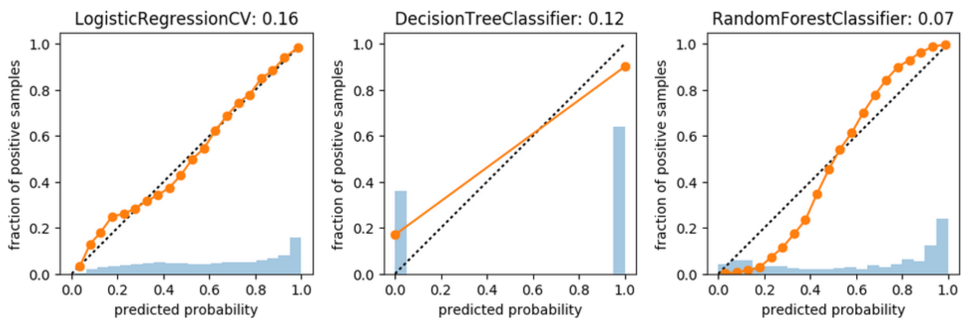
Comparing Models



Brier Score (for binary classification)

- "mean squared error of probability estimate"

$$BS = \frac{1}{n}\sum_{t=1}^{n}(\hat{p}(y) - y)^2$$



Fixing it: Calibrating a classifier

- Build another model, mapping classifier probabilities to better probabilities!
- 1d model! (or more for multi-class)

$$f_{\text{callib}}(s(\mathbf{x})) \approx p(y)$$

- s(x) is score given by model, usually $\hat{p}(y)$
- Can also work with models that don't even provide probabilities!
- Need model for f_{callib}, need to decide what data to train it on.
- Can train on training set → Overfit
- Can train using cross-validation → use data, slower

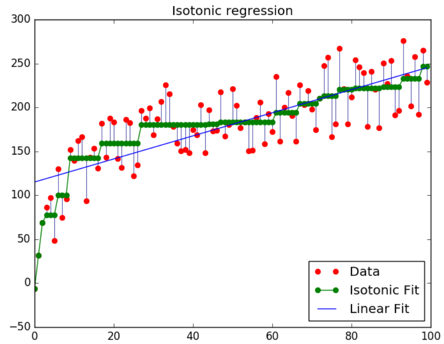
Platt Scaling

- Use a logistic sigmoid for f_{callib}
- Basically learning a 1d logistic regression (+ some tricks)
- Works well for SVMs

$$f_{\text{platt}} = \frac{1}{1 + \exp(-s(\mathbf{x}))}$$

Isotonic Regression

- Very flexible way to specify f_{callib}
- Learns arbitrary monotonically increasing step-functions in 1d.
- Groups data into constant parts, steps in between.
- Optimum monotone function on training data (wrt mse).



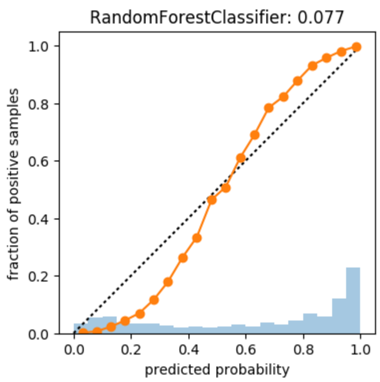
Building the model

- Using the training set is bad
- Either use hold-out set or cross-validation
- Cross-validation can be use as in stacking to make unbiased probability predictions, use that as training set.

CalibratedClassifierCV

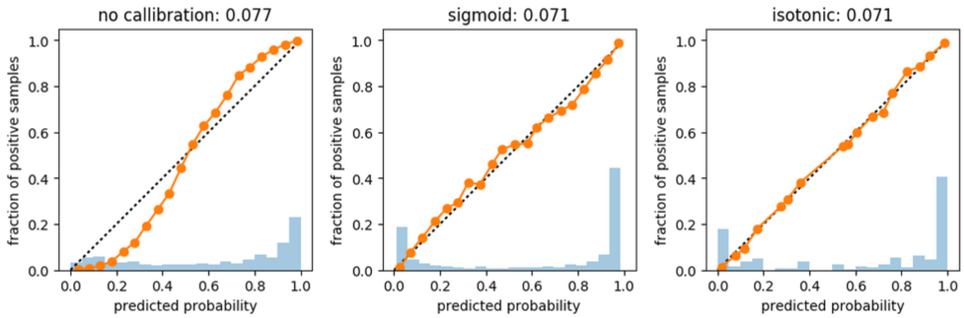
```
from sklearn.calibration import CalibratedClassifierCV
X_train_sub, X_val, y_train_sub, y_val = train_test_split(X_train, y_train,
                                                          stratify=y_train, random_state=0)
```

```
rf = RandomForestClassifier(n_estimators=100).fit(X_train_sub, y_train_sub)
scores = rf.predict_proba(X_test)[:, 1]
```




```
cal_rf = CalibratedClassifierCV(rf, cv="prefit", method='sigmoid')
cal_rf.fit(X_val, y_val)
scores_sigm = cal_rf.predict_proba(X_test)[:, 1]

cal_rf_iso = CalibratedClassifierCV(rf, cv="prefit", method='isotonic')
cal_rf_iso.fit(X_val, y_val)
scores_iso = cal_rf_iso.predict_proba(X_test)[:, 1]
```

Calibration on Random Forest



Sigmoid works well on SVM and RF which often have sigmoid shape in the calibration curve.
Isotonic is more noisy but can work in more cases.