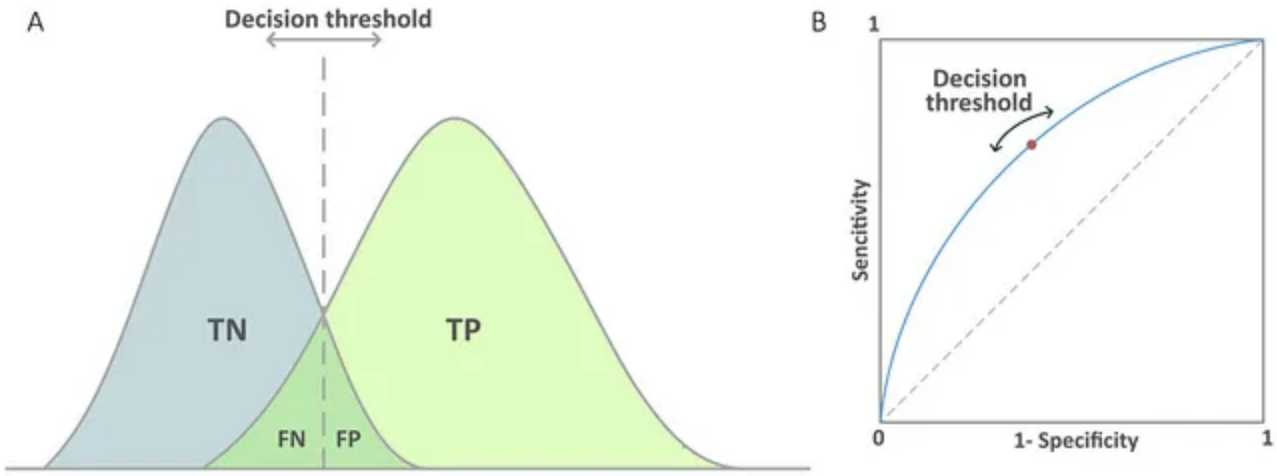
Fine tuning a classifier in scikit-learn

Kevin Arvai · Follow
Published in Towards Data Science · 6 min read · Jan 24, 2018

It's easy to understand that many machine learning problems benefit from either precision or recall as their optimal performance metric but implementing the concept requires knowledge of a detailed process. My first few attempts to fine-tune models for recall (sensitivity) were difficult, so I decided to share my experience.

This post is from my first Kaggle kernel, where my aim was not to build a robust classifier, rather I wanted to show the practicality of optimizing a classifier for sensitivity. In figure A below, the goal is to move the decision threshold to the left. This minimizes false negatives, which are especially troublesome in the dataset chosen for this post. It contains features from images of 357 benign and 212 malignant breast biopsies. A false negative sample equates to missing a diagnosis of a malignant tumor. The data file can be downloaded here.



The goal of this post is to outline how to move the decision threshold to the left in Figure A, reducing false negatives and maximizing sensitivity.

With scikit-learn, tuning a classifier for recall can be achieved in (at least) two main steps.

1. Using `GridSearchCV` to tune your model by searching for the best hyperparameters and keeping the classifier with the highest recall score.
2. Adjust the decision threshold using the precision-recall curve and the roc curve, which is a more involved method that I will walk through.

Start by loading the necessary libraries and the data.

The class distribution can be found by counting the `diagnosis` column. B for benign and M for malignant.

```
B    357
M    212
Name: diagnosis, dtype: int64
```

Convert the class labels and split the data into training and test sets. `train_test_split` with `stratify=True` results in consistent class distribution between training and test sets:

```
# show the distribution
print('y_train class distribution')
print(y_train.value_counts(normalize=True))

print('y_test class distribution')
print(y_test.value_counts(normalize=True))
```

```
y_train class distribution
0    0.626761
1    0.373239
Name: diagnosis, dtype: float64
y_test class distribution
0    0.629371
1    0.370629
Name: diagnosis, dtype: float64
```

Now that the data has been prepared, the classifier can be built.

First strategy: Optimize for sensitivity using GridSearchCV with the scoring argument.

First build a generic classifier and setup a parameter grid; random forests have many tunable parameters, which make it suitable for `GridSearchCV`. The `scorers` dictionary can be used as the `scoring` argument in `GridSearchCV`. When multiple scores are passed, `GridSearchCV.cv_results_` will return scoring metrics for each of the score types provided.

The function below uses GridSearchCV to fit several classifiers according to the combinations of parameters in the `param_grid`. The scores from `scorers` are recorded and the best model (as scored by the `refit` argument) will be selected and "refit" to the full training data for downstream use. This also makes predictions on the held out `X_test` and prints the confusion matrix to show performance.

The point of the wrapper function is to quickly reuse the code to fit the best classifier according to the type of scoring metric chosen. First, try `precision_score`, which should limit the number of false positives. This isn't well-suited for the goal of maxium sensitivity, but allows us to quickly show the difference between a classifier optimized for `precision_score` and one optimized for `recall_score`.

```
grid_search_clf = grid_search_wrapper(refit_score='precision_score')
```

```
Best params for precision_score
{'max_depth': 15, 'max_features': 20, 'min_samples_split': 3,
'n_estimators': 300}

Confusion matrix of Random Forest optimized for precision_score on
the test data:
     pred_neg  pred_pos
neg        85         5
pos         3        50
```

The precision, recall, and accuracy scores for every combination of the parameters in `param_grid` are stored in `cv_results_`. Here, a pandas DataFrame helps visualize the scores and parameters for each classifier iteration. This is included to show that although accuracy may be relatively consistent across classifiers, it's obvious that precision and recall have a trade-off. Sorting by precision, the best scoring model should be the first

record. This can be checked by looking at the parameters of the first record and comparing them to `grid_search.best_params_` above.

```
results = pd.DataFrame(grid_search_clf.cv_results_)
results = results.sort_values(by='mean_test_precision_score',
ascending=False)

results[['mean_test_precision_score', 'mean_test_recall_score',
'mean_test_accuracy_score', 'param_max_depth', 'param_max_features',
'param_min_samples_split', 'param_n_estimators']].round(3).head()
```

That classifier was optimized for precision. For comparison, to show how `GridSearchCV` selects the best classifier, the function call below returns a classifier optimized for recall. The grid might be similar to the grid above, the only difference is that the classifer with the highest recall will be refit. This will be the most desirable metric in the cancer diagnosis classification problem, there should be less false negatives on the test set confusion matrix.

```
grid_search_clf = grid_search_wrapper(refit_score='recall_score')
```

```
Best params for recall_score
{'max_depth': 5, 'max_features': 3, 'min_samples_split': 5,
'n_estimators': 100}

Confusion matrix of Random Forest optimized for recall_score on the
test data:
     pred_neg  pred_pos
neg        84         6
pos         3        50
```

Copy the same code for the generating the results table again, only this time it the best scores will be `recall`.

```
results = pd.DataFrame(grid_search_clf.cv_results_)
results = results.sort_values(by='mean_test_precision_score',
ascending=False)
```

```
results[['mean_test_precision_score', 'mean_test_recall_score',
'mean_test_accuracy_score', 'param_max_depth', 'param_max_features',
'param_min_samples_split', 'param_n_estimators']].round(3).head()
```

The first strategy doesn't yield impressive results for `recall_score`, it doesn't significantly reduce (if at all) the number of false negatives compared to the classifier optimized for `precision_score`. Ideally, when designing a cancer diagnosis test, the classifier should strive as few false negatives as possible.

Strategy 2: Adjust the decision threshold to identify the operating point

The `precision_recall_curve` and `roc_curve` are useful tools to visualize the sensitivity-specificty tradeoff in the classifier. They help inform a data scientist where to set the decision threshold of the model to maximize either sensitivity or specificity. This is called the "operating point" of the model.

> *The key to understanding how to fine tune classifiers in scikit-learn is to understand the methods* `.predict_proba()` *and* `.decision_function()`*. These*

> *return the raw probability that a sample is predicted to be in a class. This is an important distinction from the absolute class predictions returned by calling the* `.predict()` *method.*

To make this method generalizable to all classifiers in scikit-learn, know that some classifiers (like RandomForest) use `.predict_proba()` while others (like SVC) use `.decision_function()`. The default threshold for `RandomForestClassifier` is 0.5, so use that as a starting point. Create an array of the class probabilites called `y_scores`.

```
y_scores = grid_search_clf.predict_proba(X_test)[:, 1]

# for classifiers with decision_function, this achieves similar
results
# y_scores = classifier.decision_function(X_test)
```

Generate the precision-recall curve for the classifier:

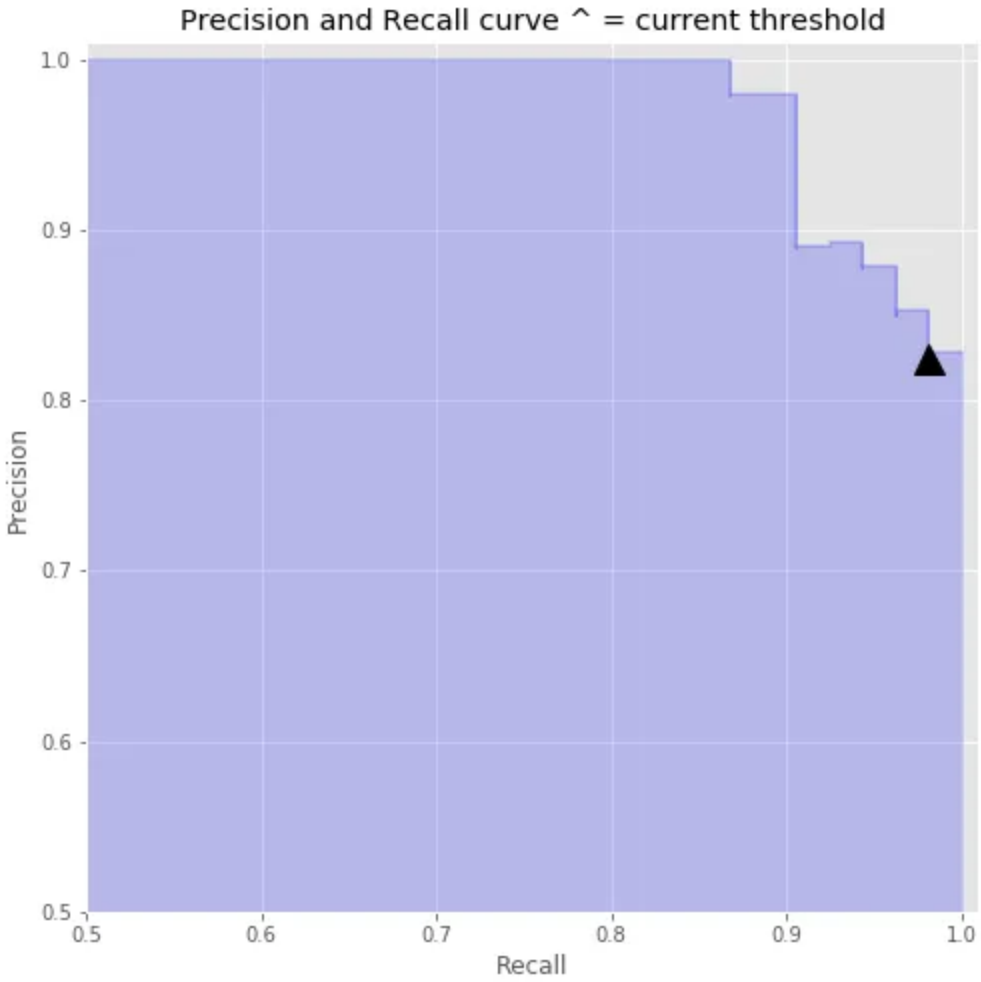
```
p, r, thresholds = precision_recall_curve(y_test, y_scores)
```

Here `adjusted_classes` is a simple function to return a modified version of `y_scores` that was calculated above, only now class labels will be assigned according to the probability threshold `t`. The other function below plots the precision and recall with respect to the given threshold value, `t`.

Re-execute this function for several iterations, changing `t` each time, to tune the threshold until there are 0 False Negatives. On this particular run, I had to go all the way down to 0.29 before reducing the false negatives to 0.

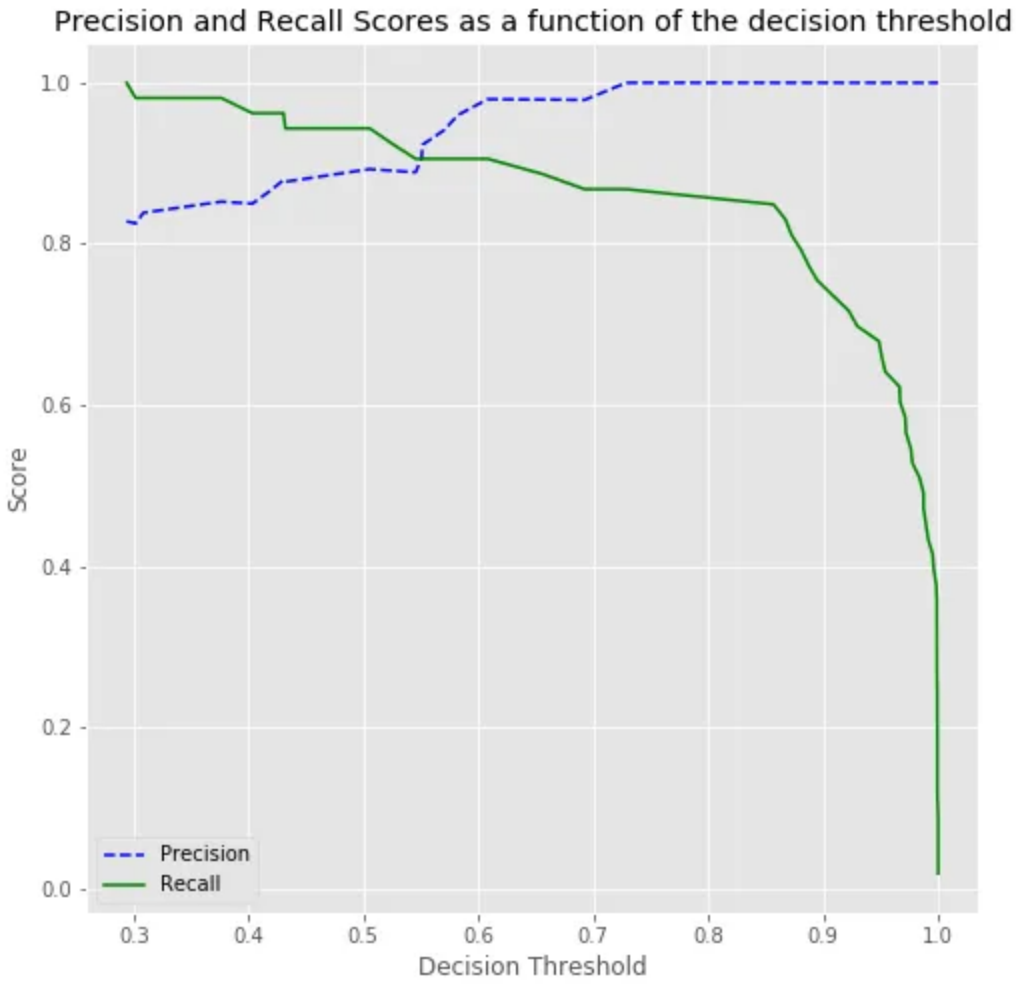
```
precision_recall_threshold(p, r, thresholds, 0.30)

pred_neg  pred_pos
neg        79        11
pos         1        52
```



Another way to view the trade off between precision and recall is to plot them together as a function of the decision threshold.

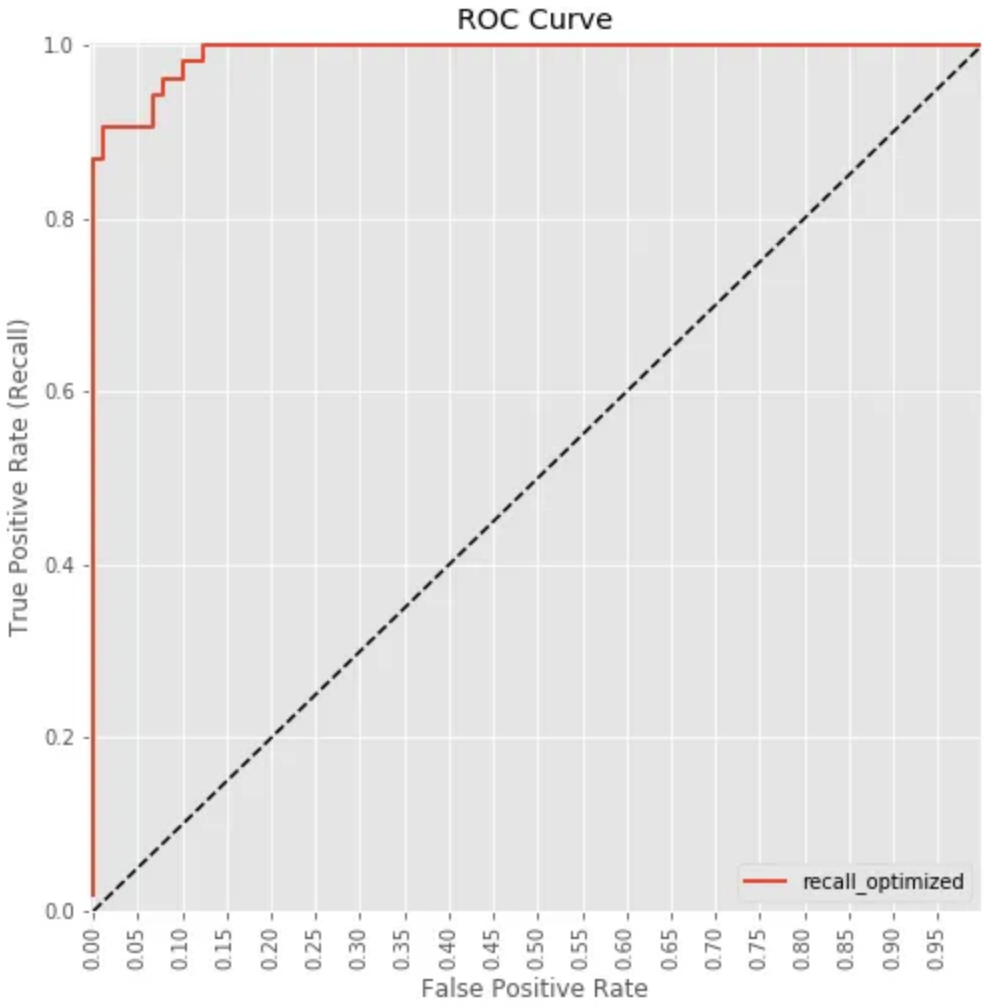
```
# use the same p, r, thresholds that were previously calculated
plot_precision_recall_vs_threshold(p, r, thresholds)
```



Finally, the ROC curve shows that to achieve a 1.0 recall, the user of the model must select an operating point that allows for some false positive rate > 0.0.

```
fpr, tpr, auc_thresholds = roc_curve(y_test, y_scores)
print(auc(fpr, tpr)) # AUC of ROC
plot_roc_curve(fpr, tpr, 'recall_optimized')
```

```
0.9914046121593292
```



Thanks for following along. The concept of tuning a model for specificity and sensitivity should be more clear and you should be comfortable implementing the methods in your scikit-learn model. I'm interested to hear suggestions to improve the code and/or the classifiers.