# Practice 7

Text classification is one of the fundamental tasks in natural language processing with broad applications such as sentiment analysis, topic labeling, spam detection, and intent detection.

There are many approaches to automatic text classification, but they all fall under three types of systems:

- Rule-based systems
- Machine learning-based systems
- Hybrid systems

## Rule-based systems

Rule-based approaches classify text into organized groups by using a set of handcrafted linguistic rules. These rules instruct the system to use semantically relevant elements of a text to identify relevant categories based on its content. Each rule consists of an antecedent or pattern and a predicted category.

Say that you want to classify news articles into two groups: *Sports* and *Politics*. First, you'll need to define two lists of words that characterize each group (e.g., words related to sports such as *football, basketball,* etc., and words related to politics, such as *prime-minister, budget, low* etc.).

Next, when you want to classify a new incoming text, you'll need to count the number of sport-related words that appear in the text and do the same for politics-related words. If the number of sports-related word appearances is greater than the politics-related word count, then the text is classified as Sports and vice versa.

Rule-based systems are human comprehensible and can be improved over time. But this approach has some disadvantages. For starters, these systems require deep knowledge of the domain. They are also time-consuming, since generating rules for a complex system can be quite challenging and usually requires a lot of analysis and testing. Rule-based systems are also difficult to maintain and don't scale well given that adding new rules can affect the results of the pre-existing rules.

## Machine learning based systems

Instead of relying on manually crafted rules, machine learning text classification learns to make classifications based on past observations. By using pre-labeled examples as training data, machine learning algorithms can learn the different associations between pieces of text, and that a particular output (i.e., tags) is expected for a particular input (i.e., text). A "tag" is the pre-determined classification or category that any given text could fall into.

The first step towards training a machine learning NLP classifier is feature extraction: a method is used to transform each text into a numerical representation in the form of a vector, like word2vec or fasttext.

Then, the machine learning algorithm is fed with training data that consists of pairs of feature sets (vectors for each text example) and tags (e.g. *sports*, *politics*) to produce a classification model.

Once it's trained with enough training samples, the machine learning model can begin to make accurate predictions. The same feature extractor is used to transform unseen text to feature sets, which can be fed into the classification model to get predictions on tags (e.g., *sports*, *politics*).

**Hybrid Systems**

Hybrid systems combine a machine learning-trained base classifier with a rule-based system, used to further improve the results. These hybrid systems can be easily fine-tuned by adding specific rules for those conflicting tags that haven't been correctly modeled by the base classifier.

**Metrics and Evaluation**

**Cross-validation** is a common method to evaluate the performance of a text classifier. It works by splitting the training dataset into random, equal-length example sets (e.g., 4 sets with 25% of the data). For each set, a text classifier is trained with the remaining samples (e.g., 75% of the samples). Next, the classifiers make predictions on their respective sets, and the results are compared against the human-annotated (actual) tags. This will determine when a prediction was right (true positives and true negatives) and when it made a mistake (false positives, false negatives).

With these results, you can build performance metrics that are useful for a quick assessment on how well a classifier works:

| | | Actual | |
|---|---|---|---|
| | | Positive | Negative |
| Predicted | Positive | TRUE POSITIVE (TP) | FALSE POSITIVE (FP) |
| | Negative | FALSE NEGATIVE (FN) | TRUE NEGATIVE (TN) |

The table above is called confusion matrix. It represents model error by different classes. For 2 classes it looks like:

| | | Positive | Negative |
|---|---|---|---|
| Predicted | Positive | 10 | 5 |
| | Negative | 0 | 15 |

For multiclass task, for example digits recognition it can look like:

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Predicted | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 |
| | 1 | 0 | 9 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| | 2 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 3 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 2 | 0 |
| | 4 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 |
| | 5 | 0 | 0 | 0 | 0 | 0 | 15 | 0 | 0 | 0 | 0 |
| | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 1 | 2 |
| | 7 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 15 | 0 | 0 |
| | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 15 | 3 |
| | 9 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 9 |

The value in every cell is equal to number of times the digit in corresponding row was classified as the digit in corresponding column. For example, the digit “4” 12 times was recognized as digit “4”. The values in the diagonal are TRUE POSITIVE values of the model for every class.

The confusion matrix allows you to understand the distribution of model responses across classes.

**Single-label metrics.** For single-label text classification, one text belongs to just one catalog, making it possible not to consider the relations among labels. Here, we introduce some evaluation metrics used for single-label text classification tasks.

**Accuracy and ErrorRate**. The Accuracy and ErrorRate are the fundamental metrics for a text classification model. The $Accuracy$ and $ErrorRate$ are respectively defined as

$$Accuracy = \frac{(TP + TN)}{N},$$

$$ErrorRate = 1 - Accuracy = \frac{(FP + FN)}{N},$$

where $N = TP + TN + FP + FN$. The main problem of this metrics is unstable result in the case of unbalanced data.

Accuracy and ErrorRate are often presented as a percentage by multiplying the result by 100.

Task 1: think, what accuracy metric equals to if the training data contain 10000 positive examples and 100 negative and the model always returns 1.

**Precision, Recall and F1.** These are vital metrics utilized for unbalanced test sets, regardless of the standard type and error rate. For example, most of the test samples have a class label. $F\ 1$ is the harmonic average of $Precision$ and $Recall$. $Precision$, $Recall,$ and $F1$ as defined:

$$Precision = \frac{TP}{TP + FP}, Recall = \frac{TP}{TP + FN},$$

$$F1 = \frac{2Precision \times Recall}{Precision + Recall}.$$

The desired results will be obtained when the accuracy, $F1$ and $Recall$ value reach 1. On the contrary, when the values become 0, the worst result is obtained. For the multi-class classification problem, the precision and recall value of each class can be calculated separately, and then the performance of the individual and whole can be analyzed.

**Multi-label metrics.** Compared with single-label text classification, multi-label text classification divides the text into multiple category labels, and the number of category labels is variable. Thus, there are some metrics designed for multi-label text classification.

***Micro–F1*** is a measure that considers the overall accuracy and recall of all labels. The $Micro{-}F1$ is defined as:

$$Micro{-}F1 = \frac{2P_t \times R_t}{P + R},$$

$$P = \frac{\sum_{t\in\mathcal{S}} TP_t}{\sum_{t\in\mathcal{S}} TP_t + FP_t}, \quad R = \frac{\sum_{t\in\mathcal{S}} TP_t}{\sum_{t\in\mathcal{S}} TP_t + FN_t}.$$

***Macro–F1*** calculates the average $F1$ of all labels. Unlike $Micro{-}F1$, which sets even weight to every example, $Macro{-}F1$ sets the same weight to all labels in the average process. Formally, $Macro{-}F1$ is defined as:

$$Macro{-}F1 = \frac{1}{\mathcal{S}} \sum_{t\in\mathcal{S}} \frac{2P_t \times R_t}{P_t + R_t},$$

$$P_t = \frac{TP_t}{TP_t + FP_t}, \quad R_t = \frac{TP_t}{TP_t + FN_t}.$$

**Calculate accuracy**

This is the case of a 1:100 imbalance with 100 and 10,000 examples respectively, and a model predicts 95 true positives, five false negatives, and 55 false positives.

```
# calculates accuracy for 1:100 dataset with 95tp, 5fn, 55fp
from sklearn.metrics import f1_score

# define actual
act_pos = [1 for _ in range(100)]
act_neg = [0 for _ in range(10000)]
y_true = act_pos + act_neg

# define predictions
pred_pos = [0 for _ in range(5)] + [1 for _ in range(95)]
pred_neg = [1 for _ in range(55)] + [0 for _ in range(9945)]
y_pred = pred_pos + pred_neg

# calculate prediction
accuracy = accuracy_score(y_true, y_pred, normalize=True)
print('Accuracy: %.3f' % accuracy)
```

```
0.994
```

**Calculate precision**

First, the case where there are 100 positive to 10,000 negative examples, and a model predicts 90 true positives and 30 false positives. The complete example is listed below.

```
# calculates precision for 1:100 dataset with 90 tp and 30 fp
from sklearn.metrics import precision_score

# define actual
act_pos = [1 for _ in range(100)]
act_neg = [0 for _ in range(10000)]
y_true = act_pos + act_neg

# define predictions
pred_pos = [0 for _ in range(10)] + [1 for _ in range(90)]
pred_neg = [1 for _ in range(30)] + [0 for _ in range(9970)]
y_pred = pred_pos + pred_neg

# calculate prediction
precision = precision_score(y_true, y_pred, average='binary')
print('Precision: %.3f' % precision)
```

```
Precision: 0.750
```

Next, we can use the same function to calculate precision for the **multiclass problem** with 1:1:100, with 100 examples in each minority class and 10,000 in the majority class. A model predicts 50 true positives and 20 false positives for class 1 and 99 true positives and 51 false positives for class 2.

When using the `precision_score()` function for multiclass classification, it is important to specify the minority classes via the "labels" argument and to perform set the "`average`" argument to '`micro`' to ensure the calculation is performed as we expect.

The complete example is listed below

```
# calculates precision for 1:1:100 dataset with 50tp,20fp, 99tp,51fp
```

```
from sklearn.metrics import precision_score

# define actual
act_pos1 = [1 for _ in range(100)]
act_pos2 = [2 for _ in range(100)]
act_neg = [0 for _ in range(10000)]
y_true = act_pos1 + act_pos2 + act_neg

# define predictions
pred_pos1 = [0 for _ in range(50)] + [1 for _ in range(50)]
pred_pos2 = [0 for _ in range(1)] + [2 for _ in range(99)]
pred_neg = [1 for _ in range(20)] + [2 for _ in range(51)] + [0 for _ in
range(9929)]
y_pred = pred_pos1 + pred_pos2 + pred_neg

# calculate prediction
precision = precision_score(y_true, y_pred, labels=[1,2], average='micro')
print('Precision: %.3f' % precision)
```

```
Precision: 0.677
```

**Calculate Recall**

The recall score can be calculated using the `recall_score()` scikit-learn function. For example, we can use this function to calculate recall for the scenarios above. First, we can consider the case of a 1:100 imbalance with 100 and 10,000 examples respectively, and a model predicts 90 true positives and 10 false negatives.

The complete example is listed below.

```
# calculates recall for 1:100 dataset with 90 tp and 10 fn
from sklearn.metrics import recall_score

# define actual
act_pos = [1 for _ in range(100)]
act_neg = [0 for _ in range(10000)]
y_true = act_pos + act_neg

# define predictions
pred_pos = [0 for _ in range(10)] + [1 for _ in range(90)]
pred_neg = [0 for _ in range(10000)]
y_pred = pred_pos + pred_neg

# calculate recall
recall = recall_score(y_true, y_pred, average='binary')
print('Recall: %.3f' % recall)
```

```
Recall: 0.900
```

We can also use the recall_score() for **imbalanced multiclass classification** problems. In this case, the dataset has a 1:1:100 imbalance, with 100 in each minority class and 10,000 in the majority class. A model predicts 77 true positives and 23 false negatives for class 1 and 95 true positives and five false negatives for class 2.

The complete example is listed below

```
from sklearn.metrics import recall_score

# define actual
act_pos1 = [1 for _ in range(100)]
```

```
act_pos2 = [2 for _ in range(100)]
act_neg = [0 for _ in range(10000)]
y_true = act_pos1 + act_pos2 + act_neg

# define predictions
pred_pos1 = [0 for _ in range(23)] + [1 for _ in range(77)]
pred_pos2 = [0 for _ in range(5)] + [2 for _ in range(95)]
pred_neg = [0 for _ in range(10000)]
y_pred = pred_pos1 + pred_pos2 + pred_neg

# calculate recall
recall = recall_score(y_true, y_pred, labels=[1,2], average='micro')
print('Recall: %.3f' % recall)
```

```
Recall: 0.860
```

**Precision vs. Recall for Imbalanced Classification**

You may decide to use precision or recall on your imbalanced classification problem.

Maximizing precision will minimize the number false positives, whereas maximizing the recall will minimize the number of false negatives.

```
Precision: Appropriate when minimizing false positives is the focus.
Recall: Appropriate when minimizing false negatives is the focus.
```

Sometimes, we want excellent predictions of the positive class. We want high precision and high recall.

**Calculate F-Measure**

The F-measure score can be calculated using the f1_score() scikit-learn function. For example, we use this function to calculate F-Measure for the scenario above. This is the case of a 1:100 imbalance with 100 and 10,000 examples respectively, and a model predicts 95 true positives, five false negatives, and 55 false positives.

The complete example is listed below.

```
# calculates f1 for 1:100 dataset with 95tp, 5fn, 55fp
from sklearn.metrics import f1_score

# define actual
act_pos = [1 for _ in range(100)]
act_neg = [0 for _ in range(10000)]
y_true = act_pos + act_neg

# define predictions
pred_pos = [0 for _ in range(5)] + [1 for _ in range(95)]
pred_neg = [1 for _ in range(55)] + [0 for _ in range(9945)]
y_pred = pred_pos + pred_neg

# calculate score
score = f1_score(y_true, y_pred, average='binary')
print('F-Measure: %.3f' % score)
```

```
F-Measure: 0.760
```

Task 2: Create one `y_true` list and `y_pred0`, `y_pred1`, `y_pred2`, `y_pred3`, according to the table below. Calculate accuracy, precision, recall and F-measure for every `y_pred_`. Paste the results to the table.

| Indexes of a list | [0..4] | [5..94] | [95..99] | [100..154] | [155..10099] |
|---|---|---|---|---|---|
| `y_true` | 1 | 1 | 1 | 0 | 0 |
| `y_pred0` | 0 | 1 | 0 | 1 | 0 |
| `y_pred1` | 1 | 0 | 1 | 0 | 1 |
| `y_pred2` | 0 | 0 | 0 | 0 | 0 |
| `y_pred3` | 1 | 1 | 1 | 1 | 1 |

Calculate accuracy, precision, recall and f-measure for every `y_pred_`. Paste the results to the table.

| | Accuracy | Precision | Recall | F–measure |
|---|---|---|---|---|
| y_pred0 | | | | |
| y_pred1 | | | | |
| y_pred2 | | | | |
| y_pred3 | | | | |

**ROC Curves and ROC–AUC**

An ROC curve (or receiver operating characteristic curve) is a plot that summarizes the performance of a binary classification model on the positive class.

The x-axis indicates the **False Positive Rate (FPR)** and the y-axis indicates the **True Positive Rate (TPR)**.

$$TPR = \frac{TP}{TP + FN}, \qquad FPR = \frac{FP}{FP + TN}$$

**TPR** shows what percentage of all positives is correctly predicted by the model.

**FPR** shows what percentage of all negatives is incorrectly predicted by the model.

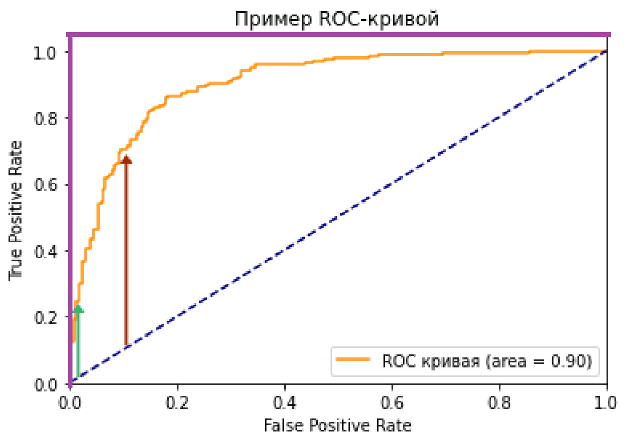
We can think of the plot as the fraction of correct predictions for the positive class (y-axis) versus the fraction of errors for the negative class (x-axis).

Ideally, we want the fraction of correct positive class predictions to be 1 (top of the plot) and the fraction of incorrect negative class predictions to be 0 (left of the plot). This highlights that the best possible classifier that achieves perfect skill is the top-left of the plot (coordinate 0,1).

- **Perfect Skill**: A point in the top left of the plot.

The **threshold** is applied to the cut-off point in probability between the positive and negative classes, which by default for any classifier would be set at 0.5, halfway between each outcome (0 and 1).

A trade-off exists between the TPR and FPR, such that **changing the threshold** of classification will change the balance of predictions towards improving the TPR at the expense of FPR, or the reverse case.



1. The ideal model looks like purple plot, that means for every positive input the output from the model is 1 and for positive input is 0, changing the threshold *t* doesn't influence on the situation at all.

2. The larger the **area** under the curve (**AUC**), the better the classification. However, if you choose a class randomly for each sample, TPR and FPR should increase at the same rate. The blue dotted line shows the TPR and FPR curve when randomly determining positive or negative for each case. For this diagonal line, the area under the curve (**AUC**) is 0.5.

3. Models represented by points below this line have worse than no skill.

4. Look at two points on the ROC curve. The green dot has a very high threshold, which means that only if you are 99% sure can you classify the case as positive. The red dot has a relatively lower threshold. This means that you can classify a case as positive if you are 90% sure.

5. How choose the optimal point on the ROC curve? It is difficult to determine the optimal point because the most appropriate threshold value must be chosen, given the scope of the model. However, the general rule is to maximize the difference (TPR-FPR), which on the graph is represented by the vertical distance between the orange and blue dotted lines.

We can plot a ROC curve for a model in Python using the roc_curve() scikit-learn function.

The function takes both the true outcomes (0,1) from the test set and the predicted probabilities for the 1 class. The function returns the false positive rates for each threshold, true positive rates for each threshold and thresholds.

```
...
# calculate roc curve
fpr, tpr, thresholds = roc_curve(testy, pos_probs)
```

Most scikit-learn models can predict probabilities by calling the *predict_proba()* function.

This will return the probabilities for each class, for each sample in a test set, e.g. two numbers for each of the two classes in a binary classification problem. The probabilities for the positive class can be retrieved as the second column in this array of probabilities.

```
...
# predict probabilities
yhat = model.predict_proba(testX)
# retrieve just the probabilities for the positive class
pos_probs = yhat[:, 1]
```

We can demonstrate this on a synthetic dataset and plot the ROC curve for a no skill classifier and a Logistic Regression model.

The make_classification() function can be used to create synthetic classification problems. In this case, we will create 1000 examples for a binary classification problem (about 500 examples per class). We will then split the dataset into a train and test sets of equal size in order to fit and evaluate the model.

```
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)

# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5,
random_state=2)
```

A Logistic Regression model is a good model for demonstration because the predicted probabilities are well-calibrated, as opposed to other machine learning models that are not developed around a probabilistic model, in which case their probabilities may need to be calibrated first (e.g. an SVM).

```
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
```

The complete example is listed below.

```
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from matplotlib import pyplot

# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)

# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5,
random_state=2)

# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)

# predict probabilities
yhat = model.predict_proba(testX)
```

```
# retrieve just the probabilities for the positive class
pos_probs = yhat[:, 1]

# plot no skill roc curve
pyplot.plot([0, 1], [0, 1], linestyle='--', label='No Skill')

# calculate roc curve for model
fpr, tpr, _ = roc_curve(testy, pos_probs)
# plot model roc curve
pyplot.plot(fpr, tpr, marker='.', label='Logistic')

# axis labels
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')

# show the legend
pyplot.legend()

# show the plot
pyplot.show()
```

Running the example creates the synthetic dataset, splits into train and test sets, then fits a Logistic Regression model on the training dataset and uses it to make a prediction on the test set.

The AUC for the ROC can be calculated in scikit-learn using the roc_auc_score() function.

Like the *roc_curve()* function, the AUC function takes both the true outcomes (0,1) from the test set and the predicted probabilities for the positive class.

```
# calculate roc auc
roc_auc = roc_auc_score(testy, pos_probs)
```

We can demonstrate this the same synthetic dataset with a Logistic Regression model.

The complete example is listed below.

```
# example of a roc auc for a predictive model
from sklearn.datasets import make_classification
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)

# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5,
random_state=2)

# no skill model, stratified random class predictions
model = DummyClassifier(strategy='stratified')
model.fit(trainX, trainy)
yhat = model.predict_proba(testX)
pos_probs = yhat[:, 1]
```

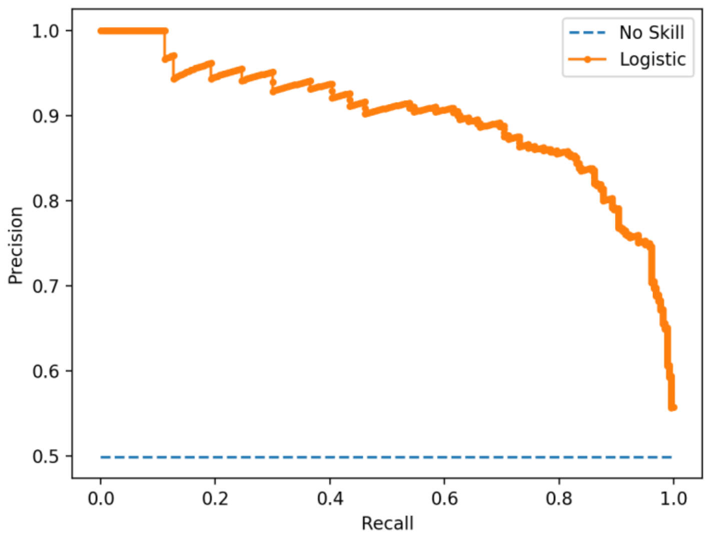
```
# calculate roc auc
roc_auc = roc_auc_score(testy, pos_probs)
print('No Skill ROC AUC %.3f' % roc_auc)

# skilled model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
yhat = model.predict_proba(testX)
pos_probs = yhat[:, 1]

# calculate roc auc
roc_auc = roc_auc_score(testy, pos_probs)
print('Logistic ROC AUC %.3f' % roc_auc)
```

```
No Skill ROC AUC 0.509
Logistic ROC AUC 0.903
```

## Precision-Recall Curves and AUC

A precision-recall curve (or PR Curve) is a plot of the precision (y-axis) and the recall (x-axis) for different probability thresholds.

A model with perfect skill is depicted as a point at a coordinate of (1,1). A skillful model is represented by a curve that bows towards a coordinate of (1,1). A no-skill classifier will be a horizontal line on the plot with a precision that is proportional to the number of positive examples in the dataset. For a balanced dataset this will be 0.5.



A precision-recall curve can be calculated in scikit-learn using the precision_recall_curve() function that takes the class labels and predicted probabilities for the minority class and returns the precision, recall, and thresholds.

```
# calculate precision-recall curve
precision, recall, _ = precision_recall_curve(testy, pos_probs)
```

The Precision-Recall AUC score can be calculated using the auc() function in scikit-learn, taking the precision and recall values as arguments.

```
# calculate the precision-recall auc
auc_score = auc(recall, precision)
```

Task 3: Plot a PR curve on the same data as the ROC curve. What would be the best threshold in the case of a PR curve? Calculate PR-AUC.