Lecture 3: Model Selection

Joaquin Vanschoren, Eindhoven University of Technology

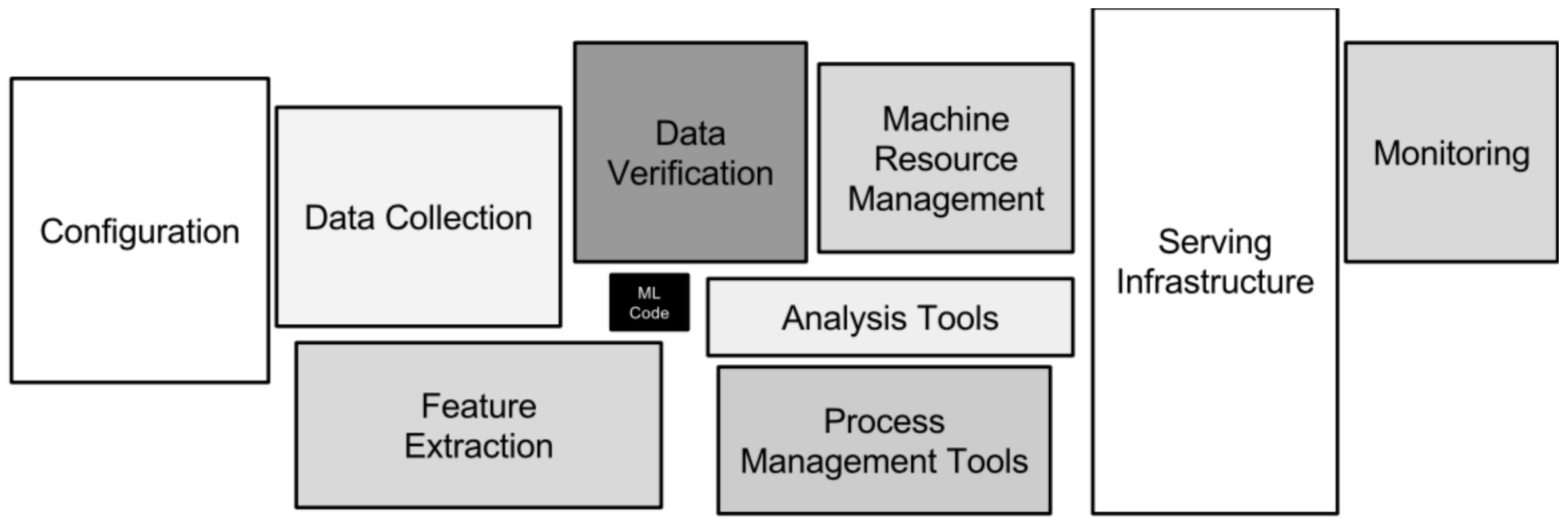
Evaluation

- To know whether we can *trust* our method or system, we need to evaluate it.
- If you cannot measure it, you cannot improve it.
- Model selection: choose between different models in a data-driven way.
- Convince others that your work is meaningful
    - Peers, leadership, clients, yourself(!)
- Keep evaluating relentlessly, adapt to changes

Designing Machine Learning systems

- Just running your favourite algorithm is usually not a great way to start
- Consider the problem at large
    - Do you want to understand phenomena or do black box modelling?
    - How to define and measure success? Are there costs involved?
    - Do you have the right data? How can you make it better?
- Build prototypes early-on to evaluate the above.

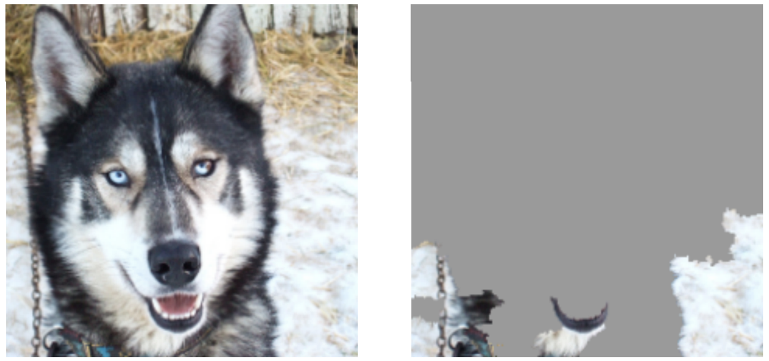
- Analyze your model's mistakes
  - Should you collect more, or additional data?
  - Should the task be reformulated?
  - Often a higher payoff than endless finetuning
- Technical debt: creation-maintenance trade-off
  - Very complex machine learning systems are hard/impossible to put into practice
  - See 'Machine Learning: The High Interest Credit Card of Technical Debt'



Only a small fraction of real-world ML systems is composed of the ML code

Real world evaluations

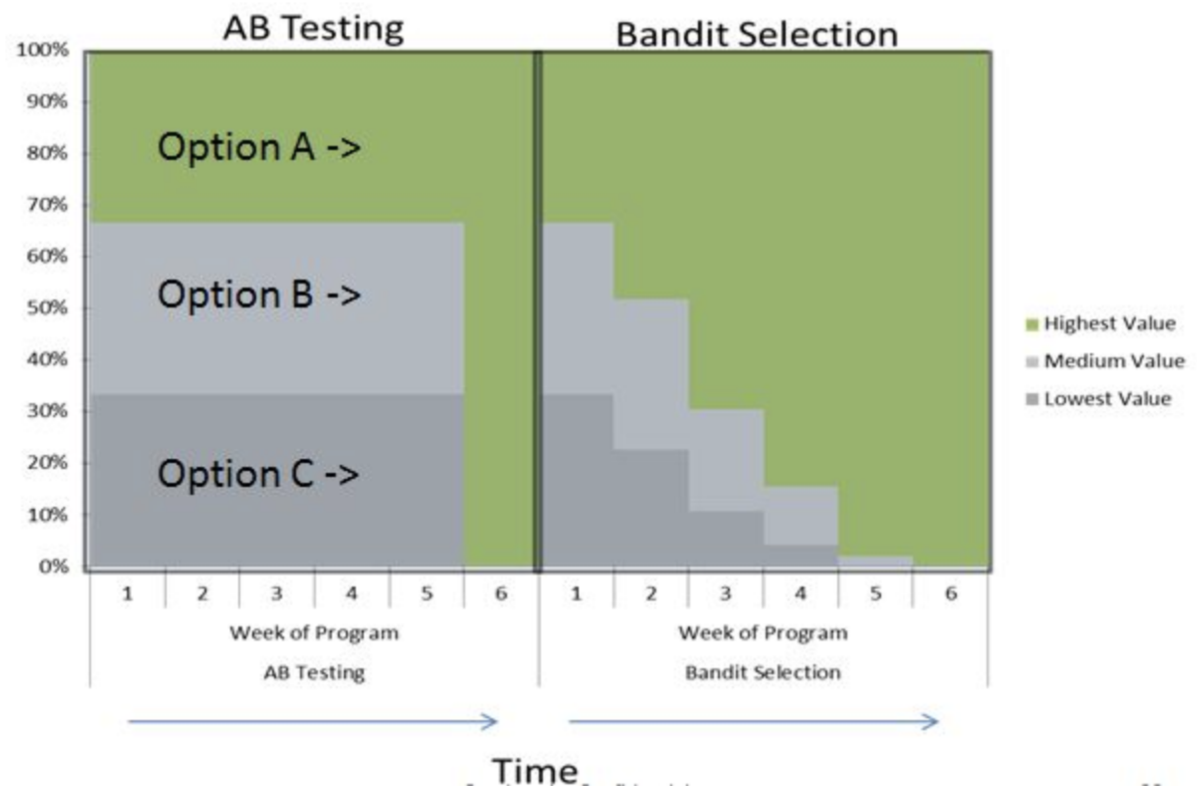
- Evaluate predictions, but also how outcomes improve *because of them*
- Feedback loops: predictions are fed into the inputs, e.g. as new data, invalidating models
- The signal your model found may just be an artifact of your biased data
    - When possible, try to *interpret* what your model has learned
    - See 'Why Should I Trust You?' by Marco Ribeiro et al.


(a) Husky classified as wolf (b) Explanation

- Adversarial situations (e.g. spam filtering) can subvert your predictions
- Do A/B testing (or bandit testing) to evaluate algorithms in the wild

A/B and bandit testing

- Test a single innovation (or choose between two models)
- Have most users use the old system, divert small group to new system
- Evaluate and compare performance
- Bandit testing: smaller time intervals, direct more users to currently winning system



Performance estimation techniques

- We do not have access to future observations
- Evaluate models *as if they are predicting the future*
- Set aside data for objective evaluation
    - How?

The holdout (simple train-test split)

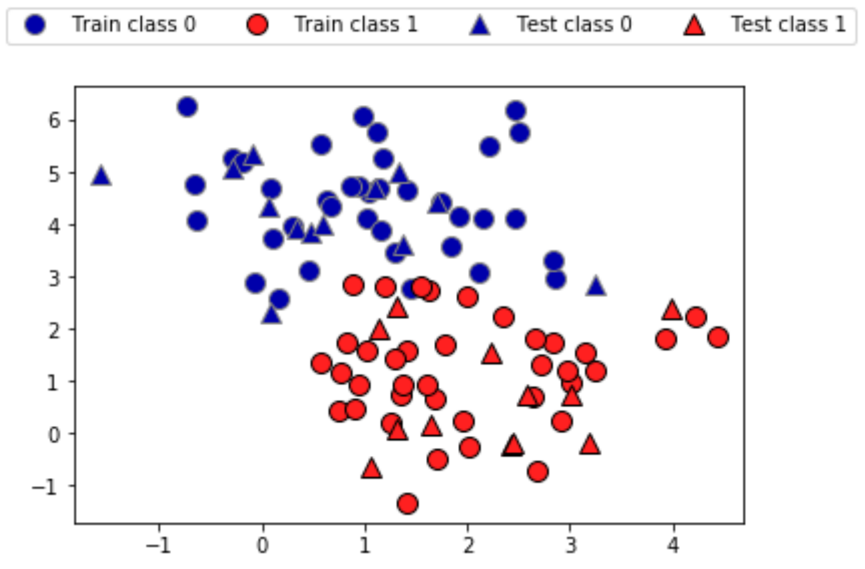
We've already seen the most basic form of evaluation:

- *Randomly* split data (and corresponding labels) into training and test set (75%-25%)
- Train (fit) a model on the training data
- Score a model on the test data (comparing predicted and true labels)
    - We are interested in how well the model *generalizes* to new (test) data

In scikit-learn: `train_test_split`

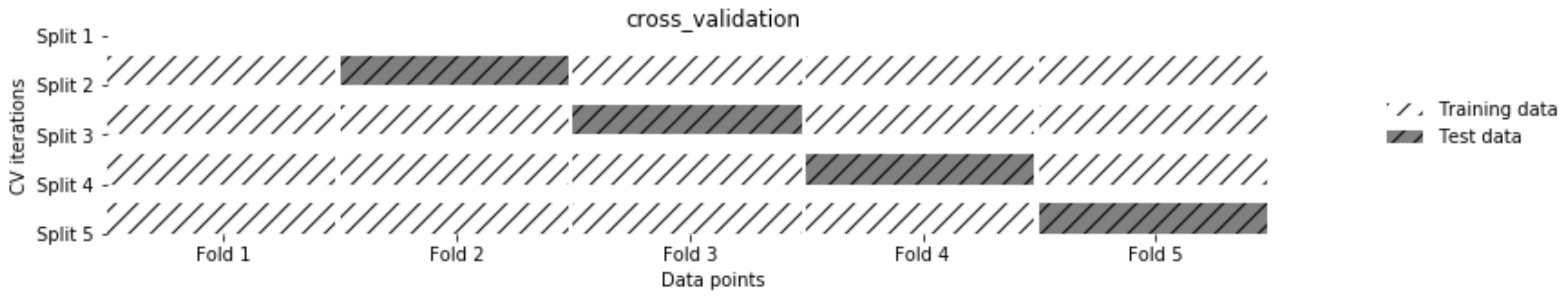
```
X, y = make_blobs(centers=2, random_state=0)
# split data and labels into a training and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0
)
# Fit a model to the training set
model = LogisticRegression().fit(X_train, y_train)
# Evaluate on the test data
test_score = model.score(X_test, y_test)
```

```
Test set score: 0.92
```



Cross-validation

- What if one random split yields different models (and scores) than another?
- Reduce bias by testing on every point exactly once
- *k-fold cross-validation* (CV): split (randomized) data into *k* equal-sized parts, called *folds*
    - First, fold 1 is the test set, and folds 2-5 comprise the training set
    - Then, fold 2 is the test set, folds 1,3,4,5 comprise the training set
    - Compute *k* evaluation scores, aggregate afterwards (e.g. take the mean)



In scikit-learn:

- `cross_val_score` function with learner, data, labels, number of folds
- Returns list of all scores. Models are built internally, but not returned
- Defaults: 3-fold CV, accuracy (classification) or R^2 (regression)
- Note that there can be quite some *variance* in the results
    - Depends on the stability of the model and the amount of training data
    - Typically, the more training data, the more stable the models

```
logreg = LogisticRegression()
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
```

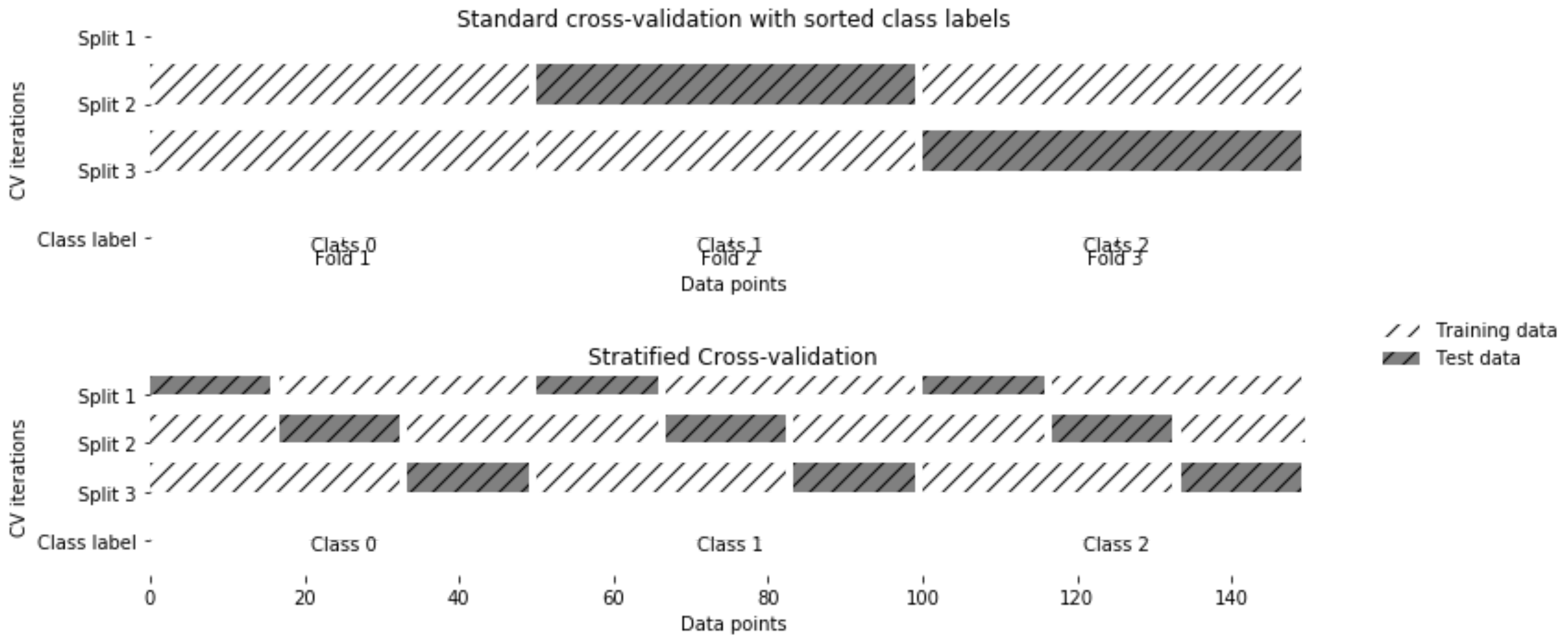
```
Cross-validation scores: [1.    0.967 0.933 0.9   1.   ]
Average cross-validation score: 0.96
Variance in cross-validation score: 0.0015
```

Benefits and drawbacks of cross-validation

- More robust: every training example will be in a test set exactly once
    - Model is evaluated on all samples, needs to do well on all
    - With a train-test split, we can be
        - 'lucky': all easy examples in test set
        - 'unlucky: all hard examples in test set
- Shows how *sensitive* the model is to the exact training set
- Better estimation of true performance
    - 10-fold CV uses 90% of all data for training (vs 75% for holdout)
    - The higher k, the more accurate the estimation
- Disadvantage: computational cost, roughly k times slower than holdout
- Unstable models (e.g. deep learning) may not converge for every fold

Stratified K-Fold cross-validation

- If the data is *unbalanced*, some classes have many fewer samples
- Likely that some classes are not present in the test set
- Stratification: *proportions* between classes are conserved in each fold
    - Order examples per class
    - Separate the samples of each class in *k* sets (strata)
    - Combine corresponding strate into folds



In scikit-learn:

- Uses stratified cross-validation by default for classification
- Normal cross-validation for regression
- Both are non-randomized (samples are not shuffled beforehand)
    - ordered data (e.g. time series) should never be randomized
- You can build folds manually with `KFold` or `StratifiedKFold`
    - randomizable (`shuffle` parameter)

```
kfold = KFold(n_splits=5, shuffle=False) # Not stratified
cross_val_score(logreg, iris.data, iris.target, cv=kfold
skfold = StratifiedKFold(n_splits=5, shuffle=True)
cross_val_score(logreg, iris.data, iris.target, cv=skfold)
```

```
Cross-validation scores KFold(n_splits=5):
[1.    0.933 0.433 0.967 0.433]
Cross-validation scores StratifiedKFold(n_splits=5, shuffle=True):
[1.    1.    0.867 0.967 1.   ]
```

Can you explain this result?

```python
kfold = KFold(n_splits=3)
print("Cross-validation scores KFold(n_splits=3):\n{}".format(
      cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

```
Cross-validation scores KFold(n_splits=3):
[0. 0. 0.]
```

Leave-One-Out cross-validation

- *k* fold cross-validation with *k* equal to the number of samples
- Completely unbiased (in terms of data splits), but computationally expensive
- But: generalizes *less* well towards unseen data
    - The training sets are correlated (overlap heavily)
    - Overfits on the data used for (the entire) evaluation
    - A different sample of the data can yield different results
- Recommended only for small datasets

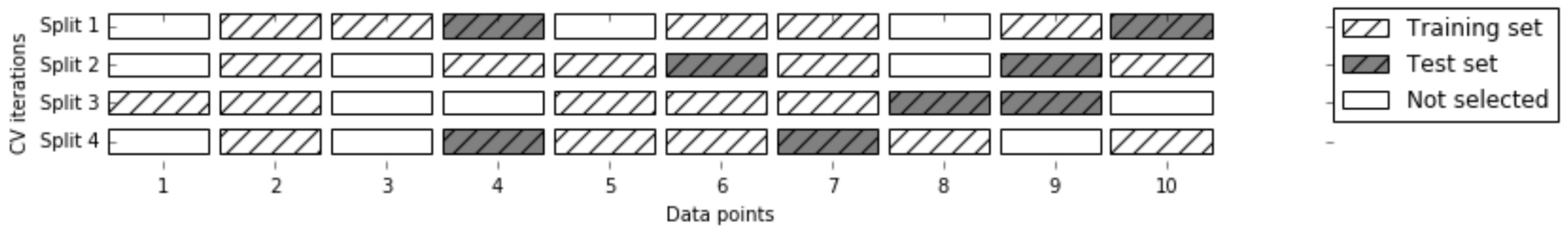
```
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
```

```
Number of cv iterations:  150
Mean accuracy: 0.95
```

Shuffle-Split cross-validation

- Samples a number of samples (`train_size`) randomly as the training set, and a disjoint number of samples (`test_size`) as the test set
- Repeat this procedure `n_iter` times, obtaining `n_iter` scores
- Handy when using very large datasets

- Example with `train_size` = 5, `test_size` = 2, `n_iter` = 4



In scikit-learn:

- `ShuffleSplit` and `StratifiedShuffleSplit` (recommended for classification)
- `train_size` and `test_size` can be absolute numbers or a percentage of the total dataset

```
shuffle_split = StratifiedShuffleSplit(test_size=.5, train_size=.5, n_sp
lits=10)
scores = cross_val_score(logreg, iris.data, iris.target, cv=shuffle_spli
t)
```

```
Cross-validation scores:
[0.907 0.96  0.92  0.96  0.973 0.96  0.973 0.96  0.92  0.96 ]
```

Note: this is related to *bootstrapping*:

- Sample n (total number of samples) data points, with replacement, as training set (the bootstrap)
- Use the unsampled (out-of-bootstrap) samples as the test set
- Repeat `n_iter` times, obtaining `n_iter` scores
- Not supported in scikit-learn, use Shuffle-Split instead.
    - With `train_size=0.66`, `test_size=0.34`
    - You can prove that bootstraps include 66% of all data points on average

Repeated cross-validation

- Cross-validation is still biased in that the initial split can be made in many ways
- Repeated, or n-times-k-fold cross-validation:
    - Shuffle data randomly, do k-fold cross-validation
    - Repeat n times, yields n times k scores
- Unbiased, very robust, but n times more expensive

Cross-validation with groups

- Sometimes the data contains inherent groups:
    - Blood analysis results on specific patients
    - Facial expressions of specific people
- With normal cross-validation, data from the same persion may end up in the training *and* test set
- We want to measure how well the model generalizes to *other* people
- We want to make sure that data points fom one person are in *either* the training or test set
    - This is called *grouping* or *blocking*
    - Leave-one-subject-out cross-validation: create test set for each user individually

In scikit-learn

- Add an array with group membership to `cross_val_scores`
- Use `GroupKFold` with the number of groups as CV procedure

```
groups = [0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3]
scores = cross_val_score(logreg, X, y, groups, cv=GroupKFold(n_splits=4
))
```

```
cross_val_score(logreg, X, y, groups, cv=GroupKFold(n_splits=4)
Cross-validation scores :
[0.667 0.667 1.    0.667]
```

Choosing a performance estimation procedure

No strict rules, only guidelines:

- Always use stratification for classification
- Use holdout for very large datasets (e.g. >1.000.000 examples)
    - Or when learners don't always converge (e.g. deep learning)
- Choose *k* depending on dataset size and resources
    - Use leave-one-out for small datasets (e.g. <500 examples)
    - Use cross-validation otherwise
        - Most popular (and theoretically sound): 10-fold CV
        - Literature suggests 5x2-fold CV is better
- Use grouping or leave-one-subject-out for grouped data

Evaluation Metrics and scoring

Keep the end-goal in mind

Evaluation vs Optimization

- Each algorithm optimizes a given objective function (on the training data)
    - E.g. remember L2 loss in Ridge regression

$$\mathcal{L}_{ridge} = \sum_i (y_i - \sum_j x_{i,j} w_j)^2 + \alpha \sum_i w_i^2$$

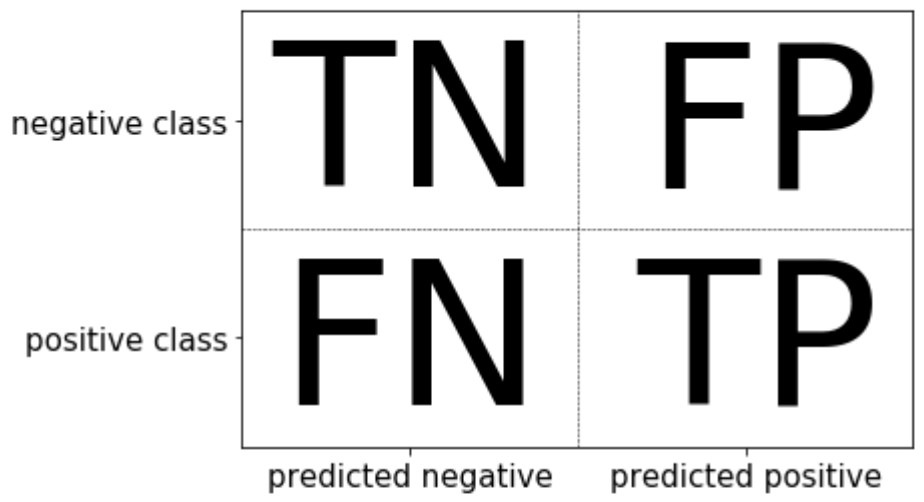
- The choice of function is limited by what can be efficiently optimized
    - E.g. gradient descent requires a differentiable loss function
- We *evaluate* the resulting model with a score that makes sense in the real world
    - E.g. percentage of correct predictions (on a test set)
- We also tune the algorithm's hyperparameters to maximize that score

Binary classification

- We have a positive and a negative class
- 2 different kind of errors:
    - False Positive (type I error): model predicts positive while the true label is negative
    - False Negative (type II error): model predicts negative while the true label is positive
- They are not always equally important
    - Which side do you want to err on for a medical test?

Confusion matrices

- We can represent all predictions (correct and incorrect) in a confusion matrix
    - n by n array (n is the number of classes)
    - Rows correspond to true classes, columns to predicted classes
    - Each entry counts how often a sample that belongs to the class corresponding to the row was classified as the class corresponding to the column.
    - For binary classification, we label these true negative (TN), true positive (TP), false negative (FN), false positive (FP)



Predictive accuracy

- Accuracy is one of the measures we can compute based on the confusion matrix:

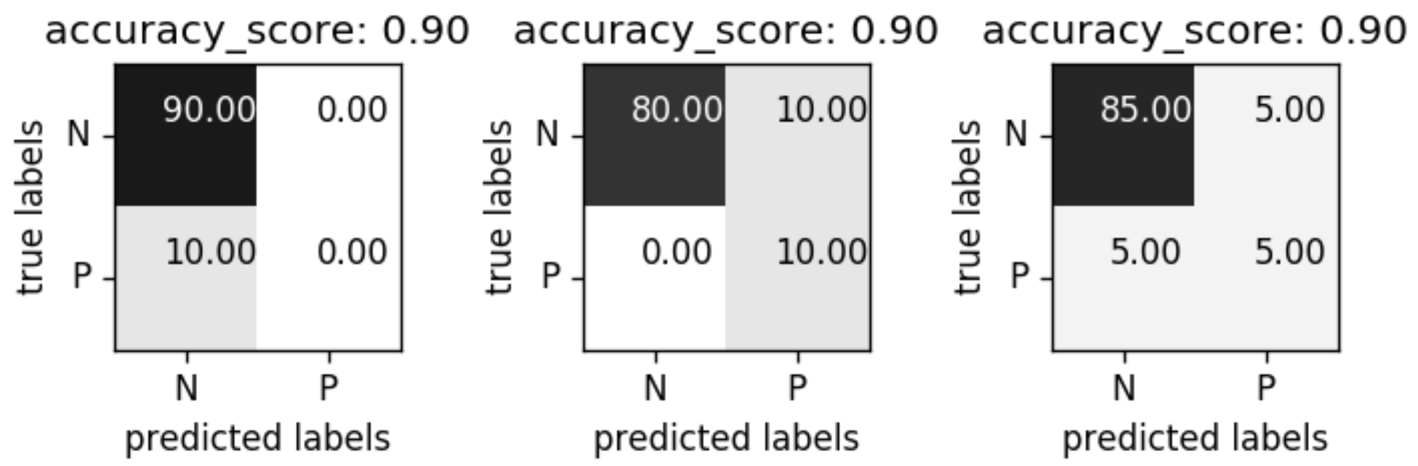
$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

- In sklearn: use `confusion_matrix` and `accuracy_score` from `sklearn.metrics`.
- Accuracy is also the default evaluation measure for classification

```
confusion_matrix(y_test, y_pred):
 [[49  4]
 [ 5 85]]
accuracy_score(y_test, y_pred):  0.9370629370629371
model.score(X_test, y_test):  0.9370629370629371
```

The problem with accuracy: imbalanced datasets

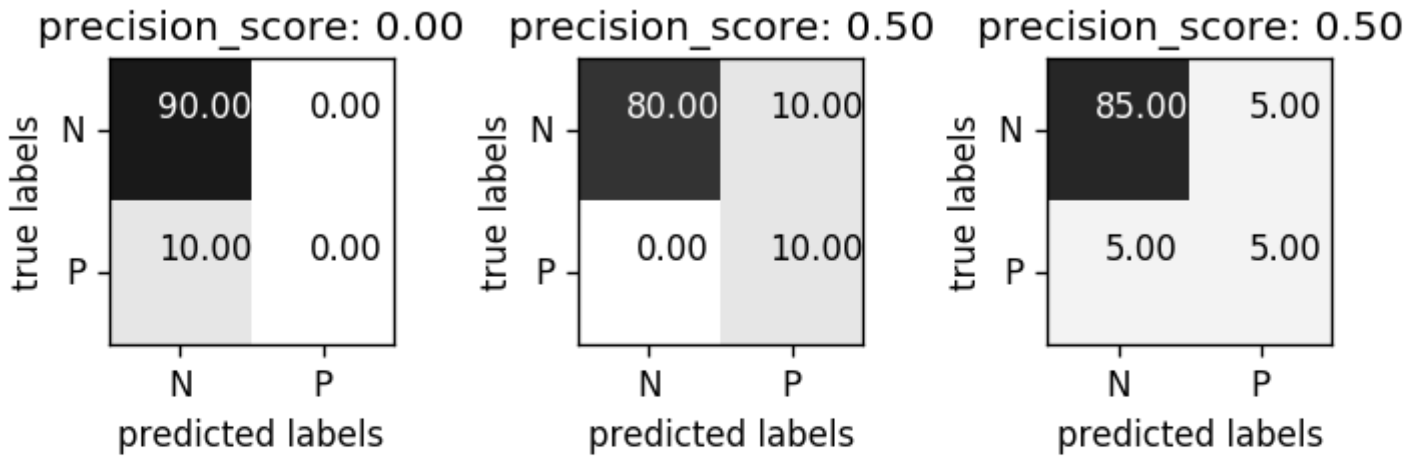
- The type of error plays an even larger role if the dataset is imbalanced
    - One class is much more frequent than the other, e.g. credit fraud
    - Is a 99.99% accuracy good enough?
- Are these three models really equally good?



Precision is used when the goal is to limit FPs

- Clinical trails: you only want to test drugs that really work
- Search engines: you want to avoid bad search results

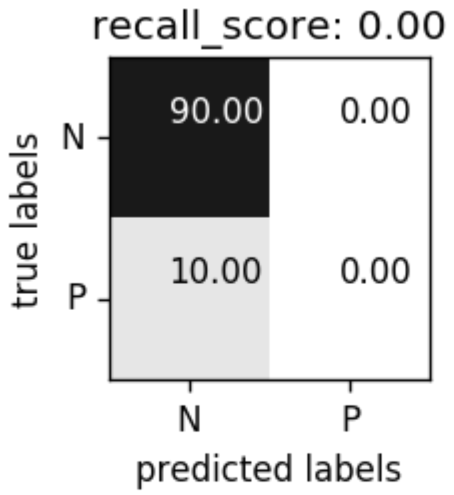
$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

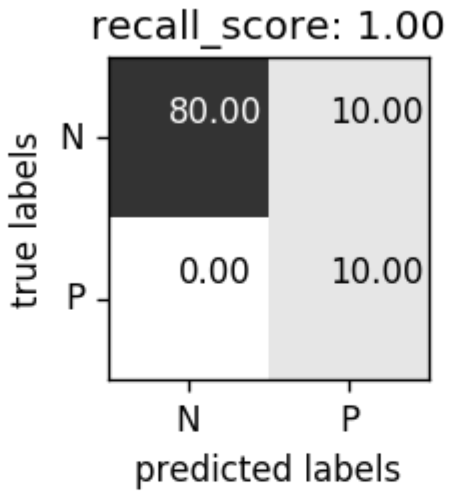


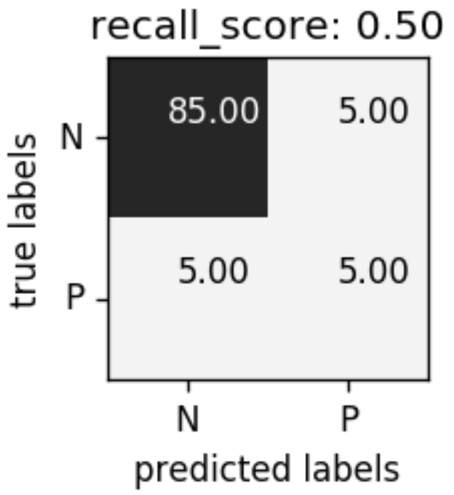
Recall is used when the goal is to limit FNs

- Cancer diagnosis: you don't want to miss a serious disease
- Search engines: You don't want to omit important hits
- Also know as sensitivity, hit rate, true positive rate (TPR)

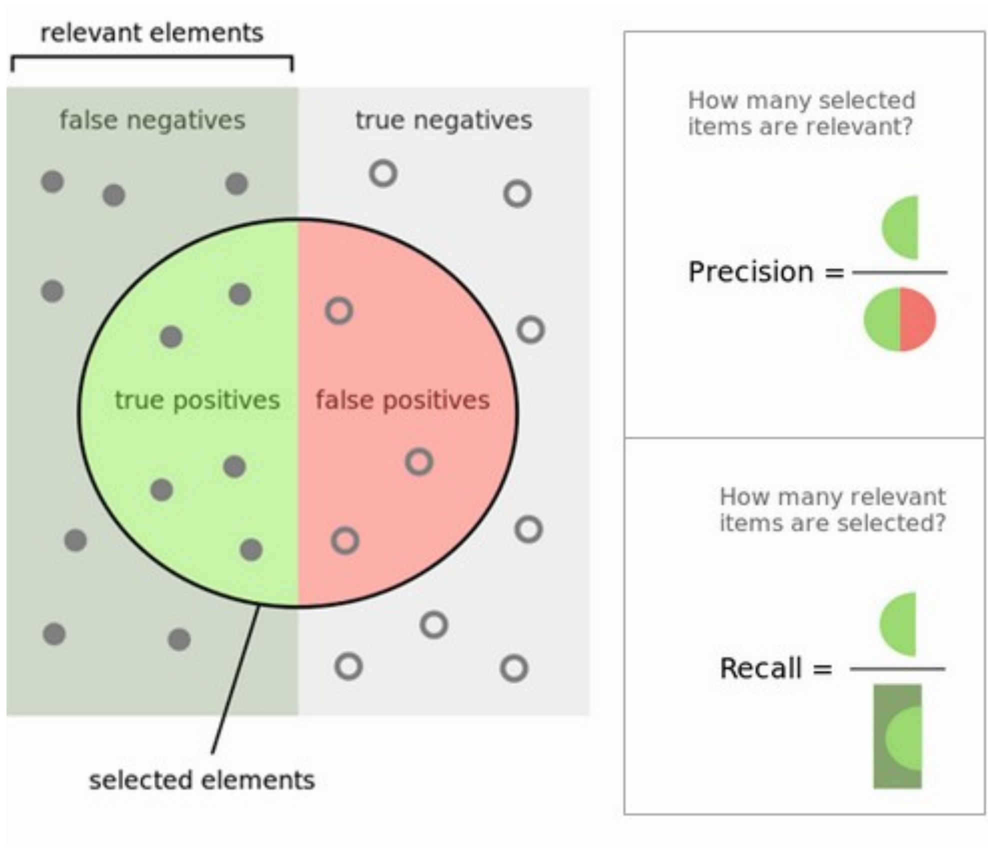
$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$





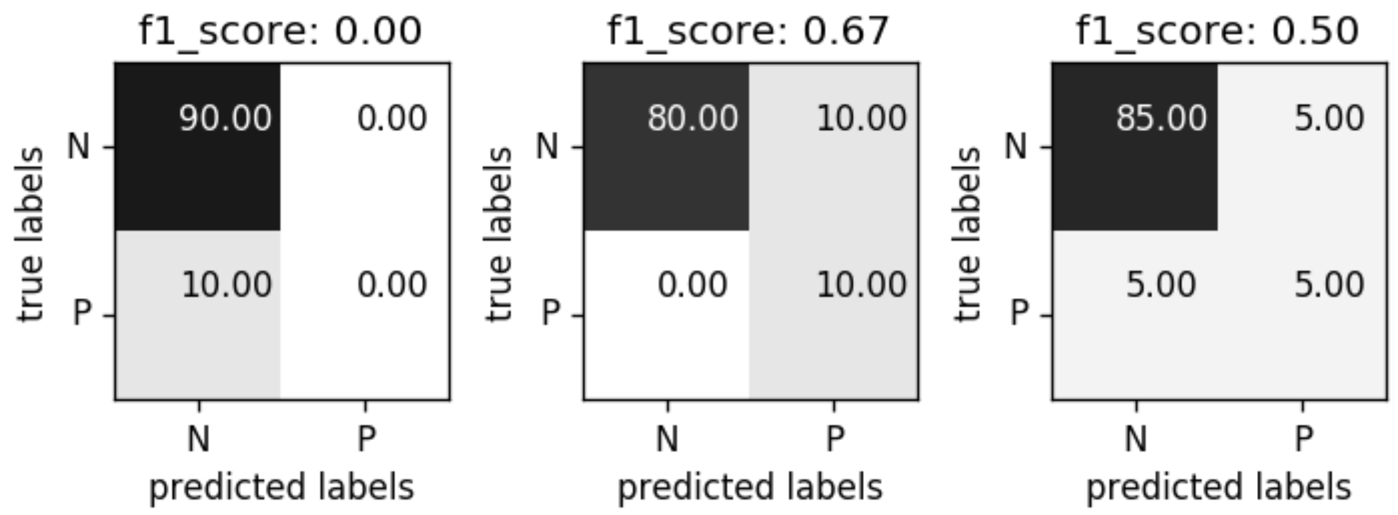


Comparison

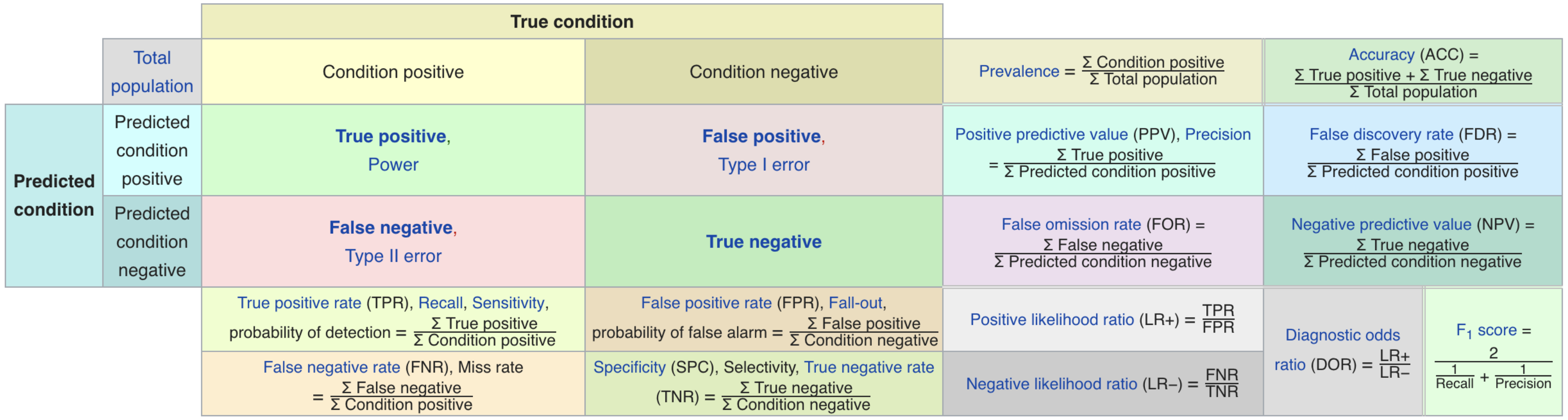

relevant elements
false negatives
true negatives
true positives
false positives
selected elements
How many selected items are relevant?
Precision =
How many relevant items are selected?
Recall =

F1-score or F1-measure trades off precision and recall:

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$



Classification measure Zoo



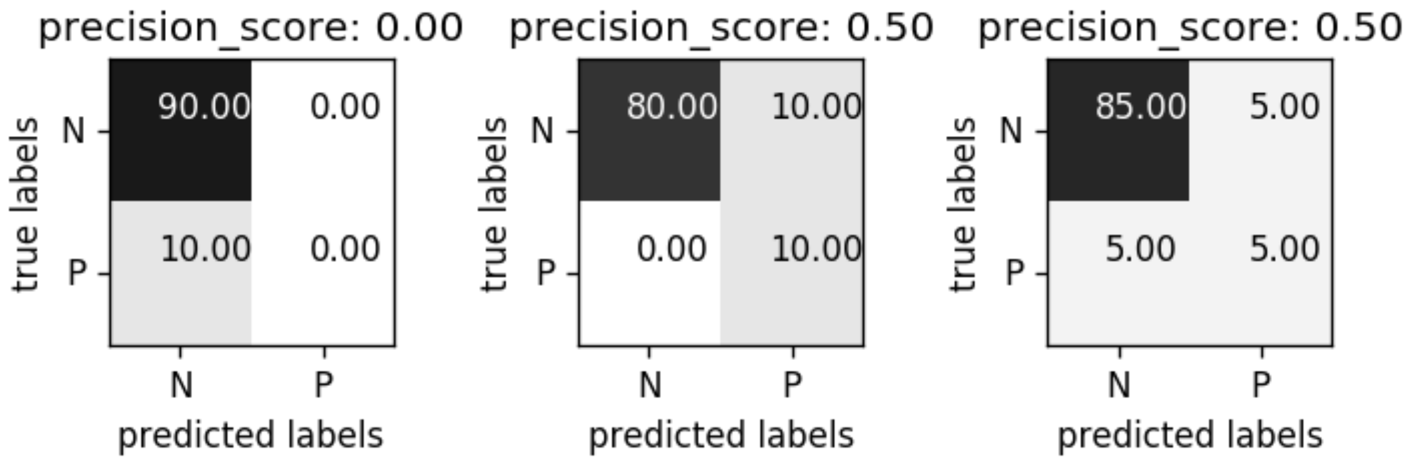
		True condition				
	Total population	Condition positive	Condition negative	Prevalence = $\frac{\Sigma \text{ Condition positive}}{\Sigma \text{ Total population}}$	Accuracy (ACC) = $\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$	
Predicted condition	Predicted condition positive	**True positive**, Power	**False positive**, Type I error	Positive predictive value (PPV), Precision $= \frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Predicted condition positive}}$	
	Predicted condition negative	**False negative**, Type II error	**True negative**	False omission rate (FOR) = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Predicted condition negative}}$	
		True positive rate (TPR), Recall, Sensitivity, probability of detection = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR+}}{\text{LR−}}$	F_1 score = $\frac{2}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}$
		False negative rate (FNR), Miss rate $= \frac{\Sigma \text{ False negative}}{\Sigma \text{ Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$	Negative likelihood ratio (LR−) = $\frac{\text{FNR}}{\text{TNR}}$		

https://en.wikipedia.org/wiki/Precision_and_recall (https://en.wikipedia.org/wiki/Precision_and_recall)"

Averaging scores per class

- Study the scores *by class* (in scikit-learn: `classification_report`)
  - One class viewed as positive, other(s) als negative
  - Support: number of samples in each class
  - Last line: weighted average over the classes (weighted by number of samples in each class)
- Averaging for scoring measure R across C classes (also for multiclass):
  - micro: count total number of TP, FP, TN, FN
  - macro
    $$\frac{1}{C}\sum_{c\in C} R(y_c, \hat{y}_c)$$
  - weighted (w_c: ratio of examples of class c)
    $$\sum_{c\in C} w_c R(y_c, \hat{y}_c)$$

Example

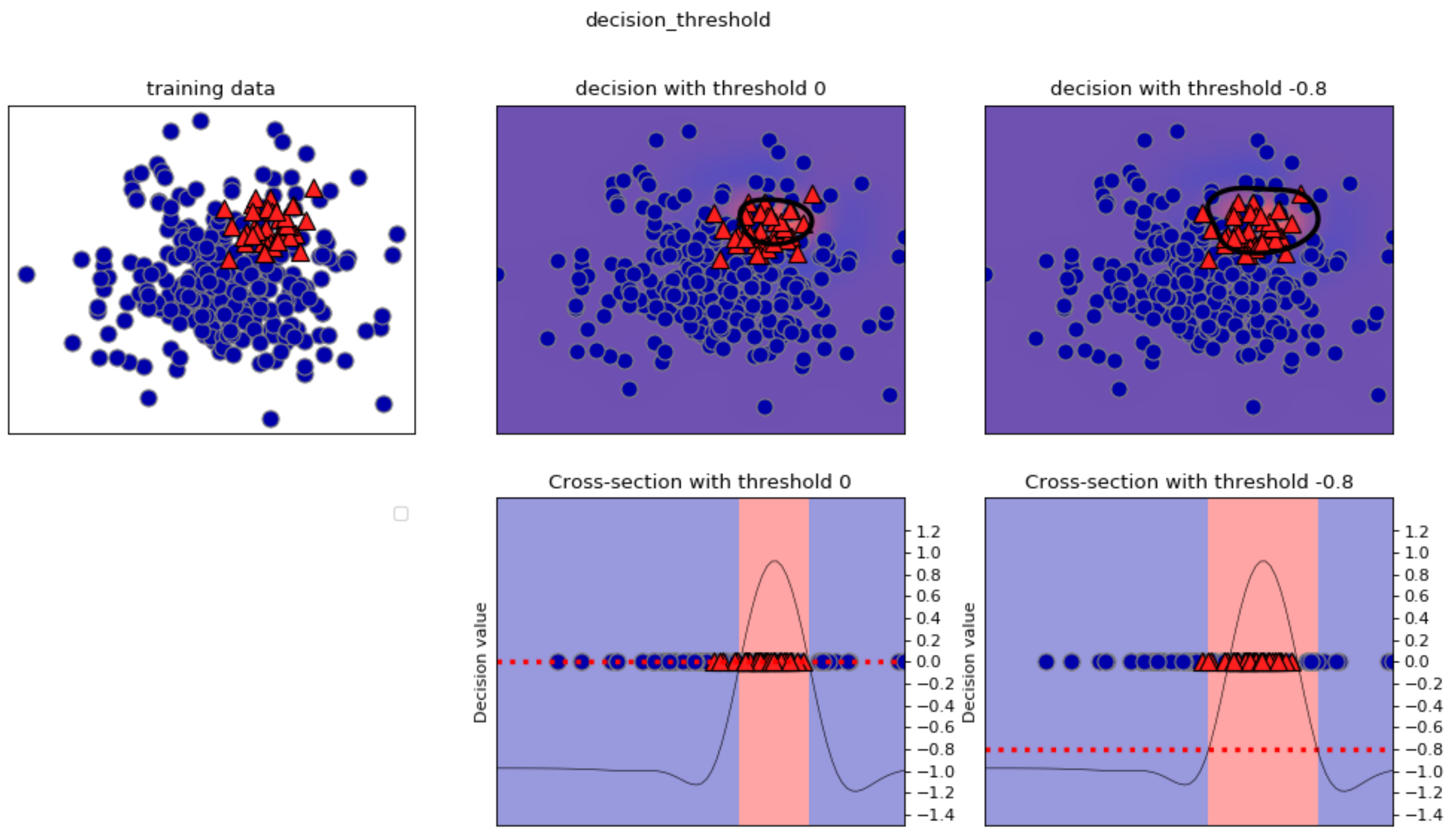

precision_score: 0.00
true labels
N
P
90.00
0.00
10.00
0.00
N
P
predicted labels
precision_score: 0.50
true labels
N
P
80.00
10.00
0.00
10.00
N
P
predicted labels
precision_score: 0.50
true labels
N
P
85.00
5.00
5.00
5.00
N
P
predicted labels

```
Matrix 1
              precision    recall  f1-score   support

           0       0.90      1.00      0.95        90
           1       0.00      0.00      0.00        10

    accuracy                           0.90       100
   macro avg       0.45      0.50      0.47       100
weighted avg       0.81      0.90      0.85       100

Matrix 2
              precision    recall  f1-score   support

           0       1.00      0.89      0.94        90
           1       0.50      1.00      0.67        10

    accuracy                           0.90       100
   macro avg       0.75      0.94      0.80       100
weighted avg       0.95      0.90      0.91       100

Matrix 3
              precision    recall  f1-score   support

           0       0.94      0.94      0.94        90
           1       0.50      0.50      0.50        10

    accuracy                           0.90       100
   macro avg       0.72      0.72      0.72       100
weighted avg       0.90      0.90      0.90       100
```

Taking uncertainty into account

- Remember that many classifiers actually return a probability per class
    - We can retrieve it with `decision_function` and `predict_proba`
- For binary classification, we threshold at 0 for `decision_function` and 0.5 for `predict_proba` by default
- However, depending on the evaluation measure, you may want to threshold differently to fit your goals
    - For instance, when a FP is much worse than a FN
    - This is called *threshold calibration*

- Imagine that we want to avoid misclassifying a positive (red) point
- Points within decision boundary (black line) are classified positive
- Lowering the decision treshold (bottom figure): fewer FN, more FP



- Studying the classification report, we see that lowering the threshold yields:
  - higher recall for class 1 (we risk more FPs in exchange for more TP)
  - lower precision for class 1
- We can often trade off precision for recall

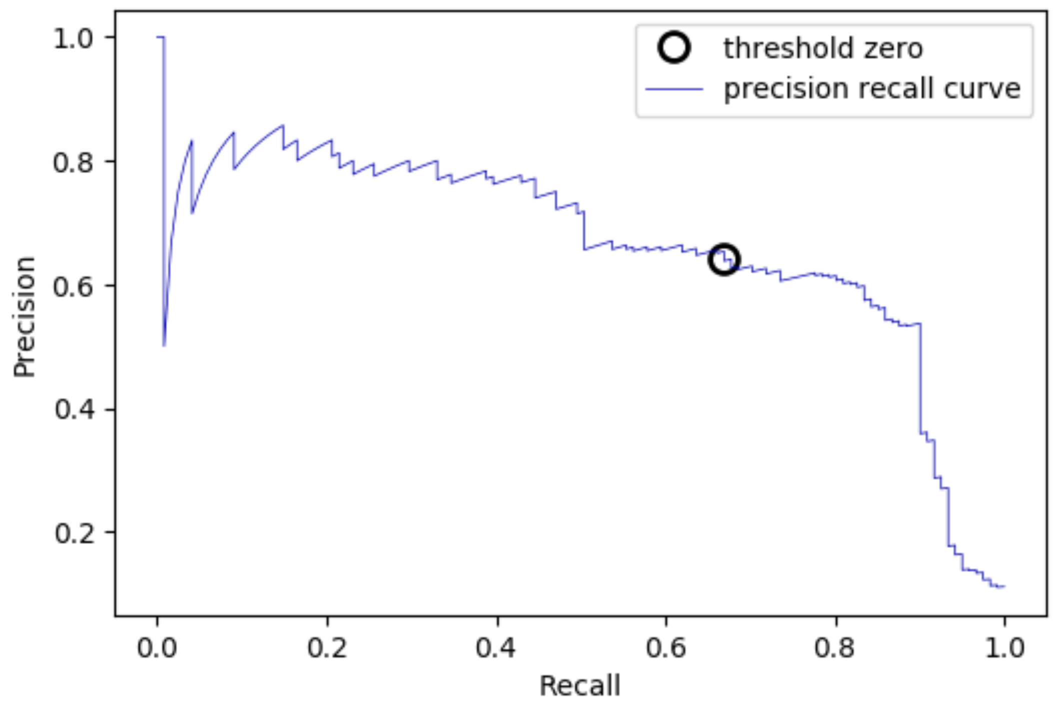
```
Threshold 0
              precision    recall  f1-score   support

           0       0.91      0.96      0.93        96
           1       0.67      0.47      0.55        17

    accuracy                           0.88       113
   macro avg       0.79      0.71      0.74       113
weighted avg       0.87      0.88      0.88       113


Threshold -0.8
              precision    recall  f1-score   support

           0       0.98      0.92      0.95        96
           1       0.65      0.88      0.75        17

    accuracy                           0.91       113
   macro avg       0.81      0.90      0.85       113
weighted avg       0.93      0.91      0.92       113
```

Precision-Recall curves

- The best threshold depends on your application, should be driven by real-world goals.
- You can have arbitrary high recall, but you often want reasonable precision, too.
- It is not clear beforehand where the optimale trade-off (or *operating point*) will be, so it is useful to look at all possible thresholds
- Plotting precision against recall for all thresholds yields a **precision-recall curve**

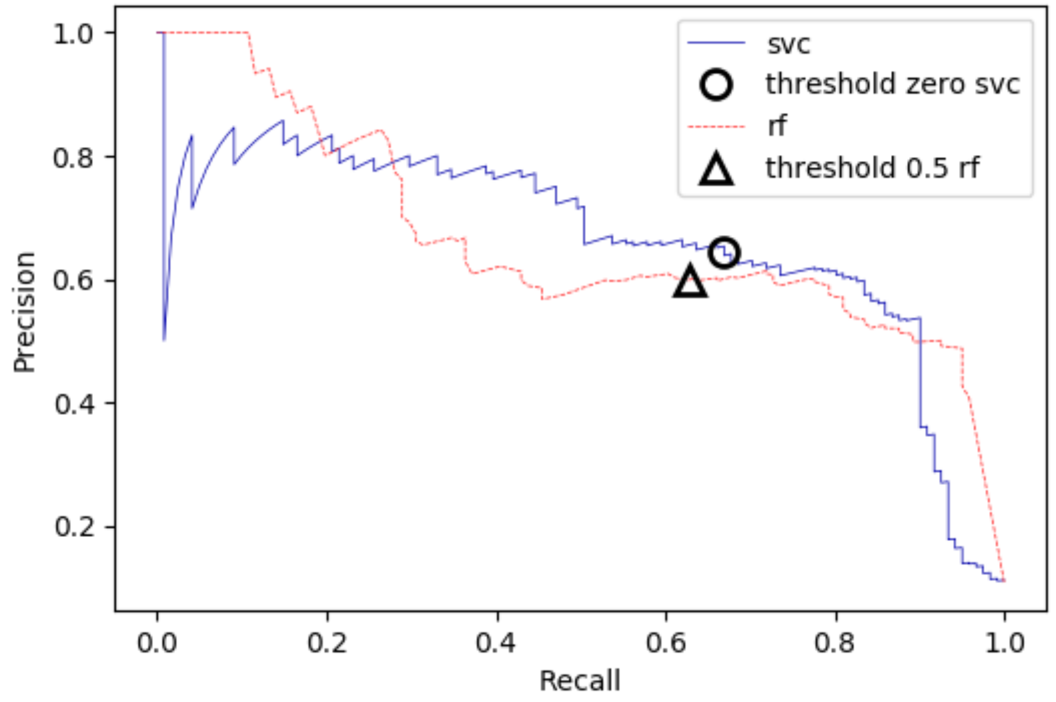
- In scikit-learn, this is included in the `sklearn.metrics` module
- Returns all precision and recall values for all thresholds
    - Vary threshold from lowest to highest decision function score in the predictions
    - Or from highest to lowest class probability

- The default tradeoff (chosen by the `predict` method) is shown as *threshold zero*.
    - Higher threshold, more precision (move left)
    - Lower threshold, more recall (move right)
- The closer the curve stays to the upper-right corner, the better
    - High precision and high recall
- Here, it is possible to still get a precision of 0.5 with high recall



Model selection

- Different classifiers work best in different parts of the curve (at different operating points)
- RandomForest (in red) performs better at the extremes, SVM better in center
- The area under the precision-recall curve (AUPRC) is often used as a general evaluation measure



Note that the F1-measure completely misses these subtleties

```
f1_score of random forest: 0.610
f1_score of svc: 0.656
```

- The area under the precision-recall curve is returned by the `average_precision_score` measure
    - It's actually a close approximation of the actual area
- This is a good automatic measure, but also hides the subtleties

```
Average precision of random forest: 0.660
Average precision of svc: 0.666
```

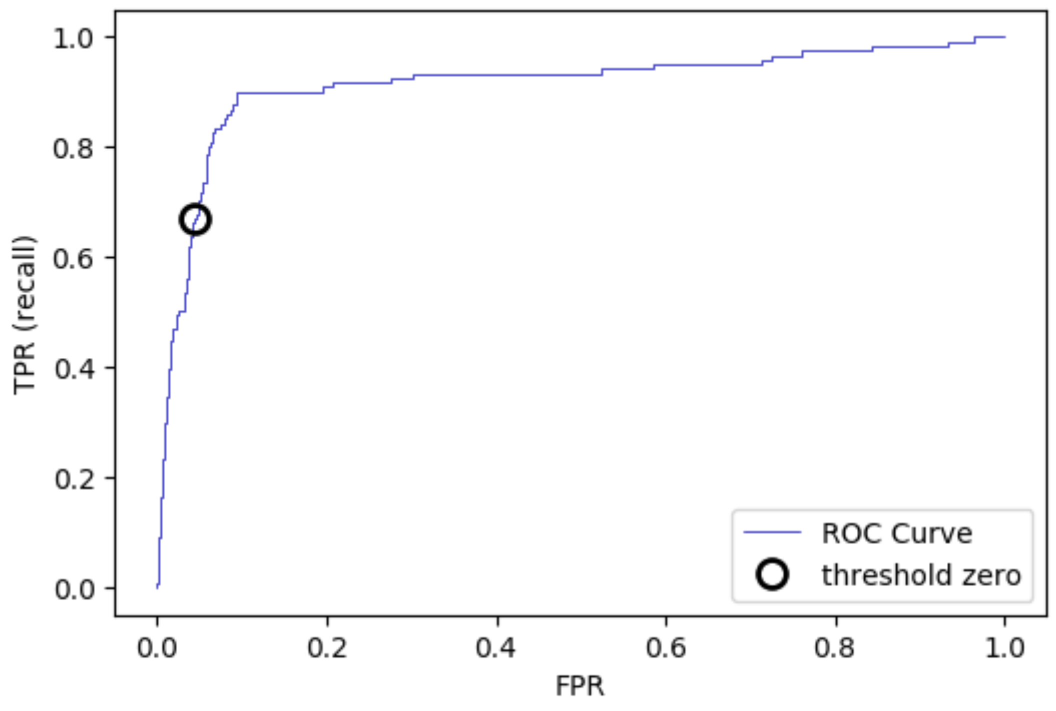
Receiver Operating Characteristics (ROC) and AUC

- There is another trade-off between recall (true positive rate, TPR) and the false positive rate (FPR).
- The 2D space created by TPR and FPR is called the Receiver Operating Characteristics (ROC) space
- A model will be at one point in this ROC space

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

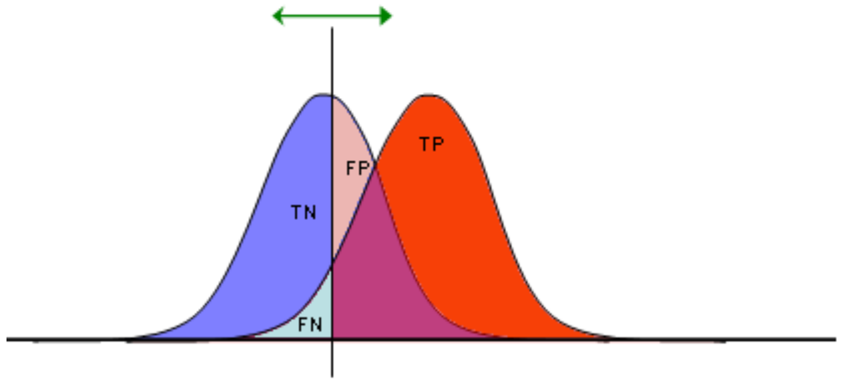
$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

- Varying the decision threshold yields the ROC curve
- It can be computed with the `roc_curve` function
    - Lower threshold, more recall/TPR, move right
    - High threshold, fewer FPs, move left
- Ideal is close to the top left: high recall, low FPR
- Inspect the curve to find the preferred calibration
    - Here, we can get much higher recall with slightly worse FPR

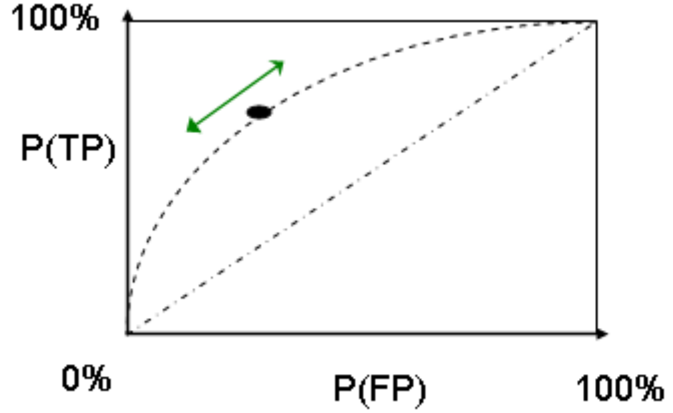


Visualization

- The blue probability density shows the probability p(x) that the model predicts blue if a data point has a certain predicted probability x to be blue. Same for red.
- In a random classifier the probability densities completely overlap.
- All points with a predicted probability higher than the threshold are predicted positive, others negative
- As we increase the threshold, we'll get fewer FPs, more FNs. We move from right to left along the ROC curve.

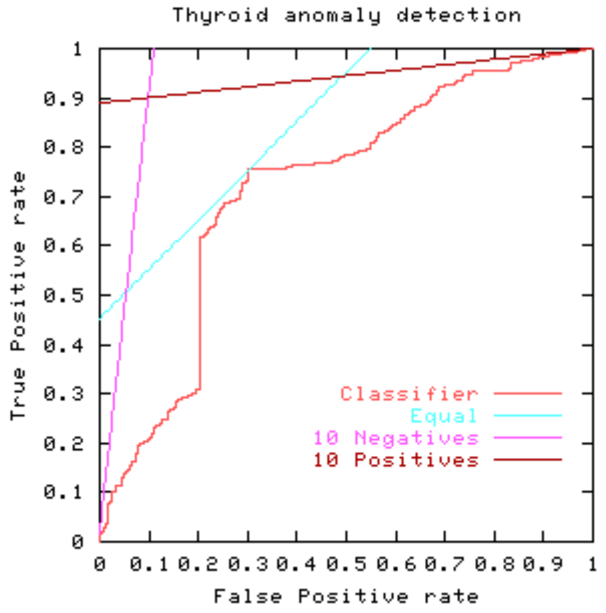


TP	FP
FN	TN
1	1



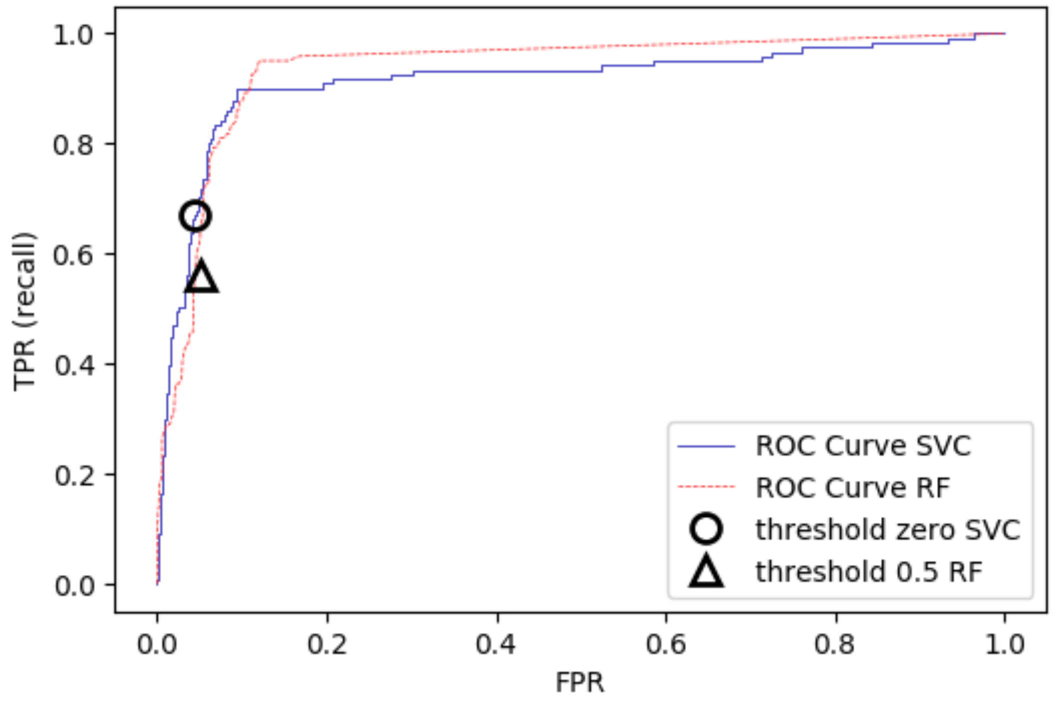
ROC Isometrics

- Different *costs* can be involved for FP and FN
- This yields different *isometrics* (lines of equal cost) in ROC space
- The optimal threshold is the point on the ROC curve where the cost in minimal
  - If a FP and FN are weigthed equally, cost lines follow the diagonal (blue line)
  - If a FP is 10 times worse than a FN: pink line
  - IF a FN is 10 times worse than a FP: red line



Model selection

- Again, we can compare multiple models by looking at the ROC curves
- We can calibrate the threshold depending on whether we need high recall or low FPR
- We can select between algorithms (or hyperparameters) depending on the involved costs.



Area under the ROC curve

- A good summary measure is the area under the ROC curve (AUROC or AUC)
- Compute using the `roc_auc_score`
    - Don't use `auc` (uses less accurate trapezoidal rule)

```
rf_auc = roc_auc_score(y_test, rf.predict_proba(X_test)[:, 1])
svc_auc = roc_auc_score(y_test, svc.decision_function(X_test))
```

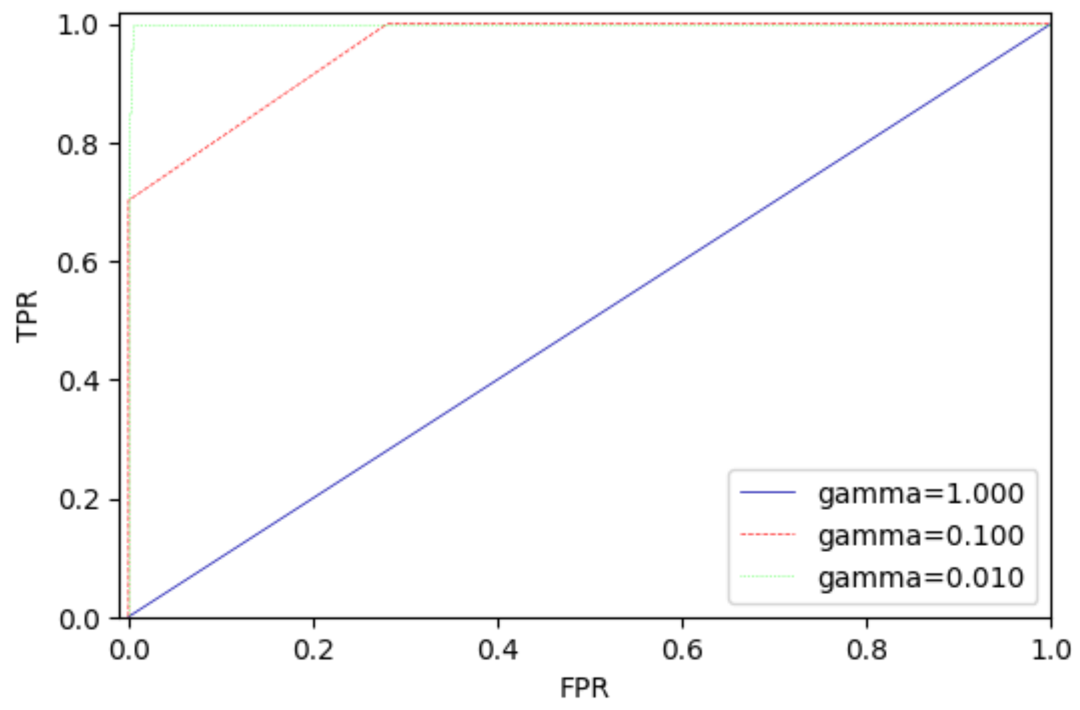
```
AUC for Random Forest: 0.937
AUC for SVC: 0.916
```

Imbalanced classes

- AUC is popular because it is insensitive to class imbalance
    - Random guessing always yields TPR=FPR
    - All points are on the diagonal line, hence an AUC of 0.5
    - Hint: use the visualization of TPR,FPR to see this

- Example: unbalanced digits
  - 3 models, ACC is the same, AUC not
  - If we optimize for ACC, our model could be just random guessing

```
gamma = 1.000  accuracy = 0.90  AUC = 0.5000
gamma = 0.100  accuracy = 0.90  AUC = 0.9582
gamma = 0.010  accuracy = 0.90  AUC = 0.9995
```



Take home message

- AUC is highly recommended, especially on imbalanced data
- Remember to calibrate the threshold to your needs

Multi-class classification

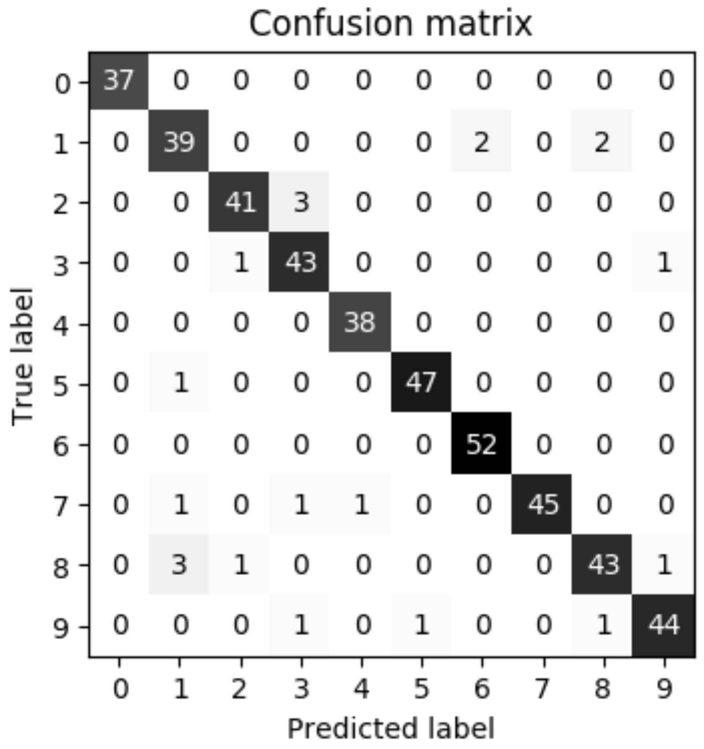
- Multiclass metrics are derived from binary metrics, averaged over all classes
- Let's consider the full (10-class) handwritten digit recognition data

Confusion matrix

```
Accuracy: 0.953
Confusion matrix:
[[37  0  0  0  0  0  0  0  0  0]
 [ 0 39  0  0  0  0  2  0  2  0]
 [ 0  0 41  3  0  0  0  0  0  0]
 [ 0  0  1 43  0  0  0  0  0  1]
 [ 0  0  0  0 38  0  0  0  0  0]
 [ 0  1  0  0  0 47  0  0  0  0]
 [ 0  0  0  0  0  0 52  0  0  0]
 [ 0  1  0  1  1  0  0 45  0  0]
 [ 0  3  1  0  0  0  0  0 43  1]
 [ 0  0  0  1  0  1  0  0  1 44]]
```

Visualized as a heatmap

- Which digits are easy to predict? Which ones are confused?



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	37	0	0	0	0	0	0	0	0	0
1	0	39	0	0	0	0	2	0	2	0
2	0	0	41	3	0	0	0	0	0	0
3	0	0	1	43	0	0	0	0	0	1
4	0	0	0	0	38	0	0	0	0	0
5	0	1	0	0	0	47	0	0	0	0
6	0	0	0	0	0	0	52	0	0	0
7	0	1	0	1	1	0	0	45	0	0
8	0	3	1	0	0	0	0	0	43	1
9	0	0	0	1	0	1	0	0	1	44

Precision, recall, F1-score now yield 10 per-class scores

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        37
           1       0.89      0.91      0.90        43
           2       0.95      0.93      0.94        44
           3       0.90      0.96      0.92        45
           4       0.97      1.00      0.99        38
           5       0.98      0.98      0.98        48
           6       0.96      1.00      0.98        52
           7       1.00      0.94      0.97        48
           8       0.93      0.90      0.91        48
           9       0.96      0.94      0.95        47

    accuracy                           0.95       450
   macro avg       0.95      0.95      0.95       450
weighted avg       0.95      0.95      0.95       450
```

Different ways to compute average

- macro-averaging: computes unweighted per-class scores: $\frac{\sum_{i=0}^{n} score_i}{n}$
  - Use when you care about each class equally much
- weighted averaging: scores are weighted by the relative size of the classes (support): $\frac{\sum_{i=0}^{n} score_i weight_i}{n}$
  - Use when data is imbalanced
- micro-averaging: computes total number of FP, FN, TP over all classes, then computes scores using these counts: $recall = \frac{\sum_{i=0}^{n} TP_i}{\sum_{i=0}^{n} TP_i + \sum_{i=0}^{n} FN_i}$
  - Use when you care about each sample equally much

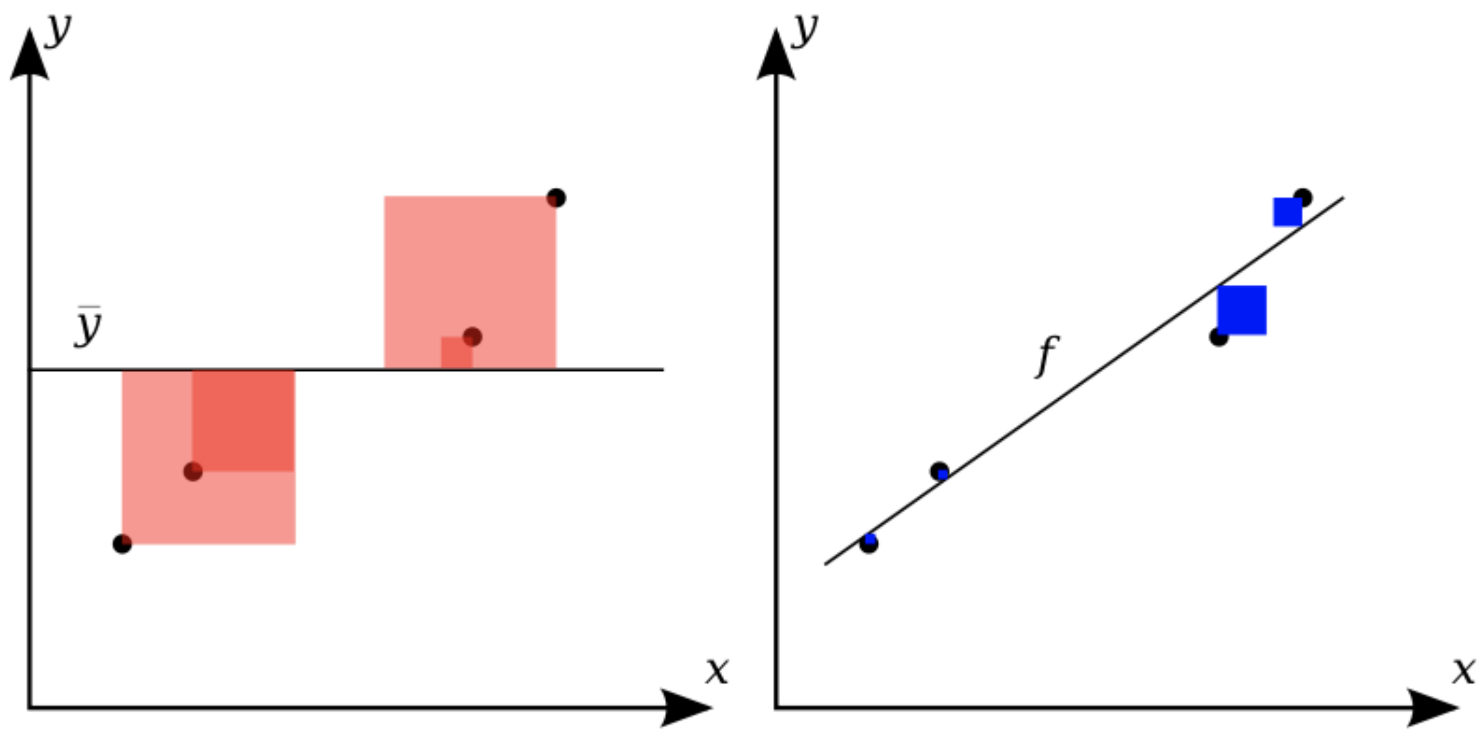
```
Micro average f1 score: 0.953
Weighted average f1 score: 0.953
Macro average f1 score: 0.954
```

Regression metrics

Most commonly used are

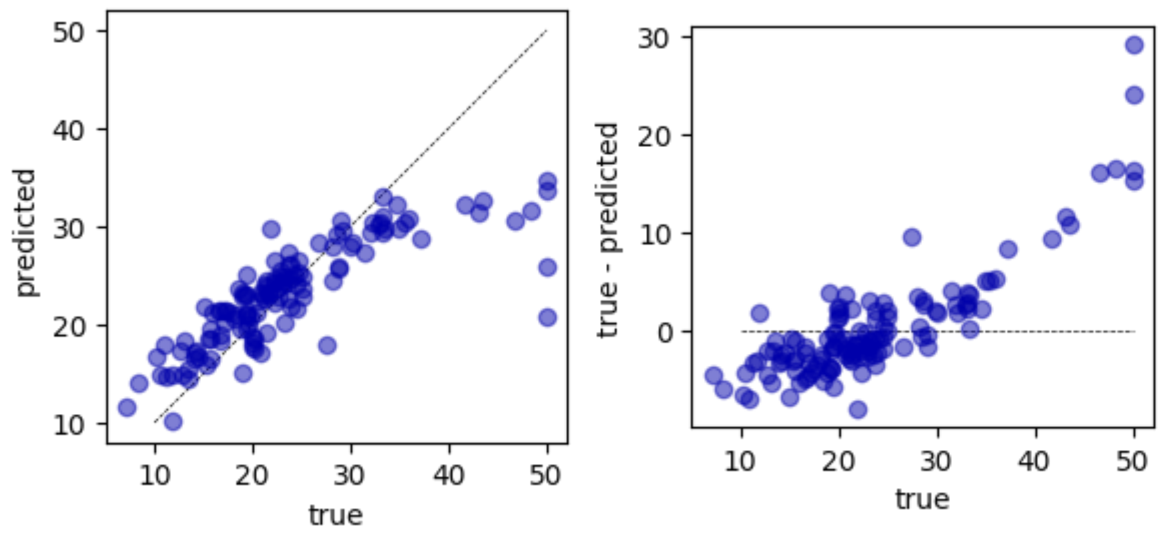
- (root) mean squared error: $\frac{\sum_i (y_{pred_i} - y_{actual_i})^2}{n}$
- mean absolute error: $\frac{\sum_i |y_{pred_i} - y_{actual_i}|}{n}$
    - Less sensitive to outliers and large errors
- R squared (r2): $1 - \frac{\sum_i (y_{pred_i} - y_{actual_i})^2}{\sum_i (y_{mean} - y_{actual_i})^2}$
    - Ratio of variation explained by the model / total variation
    - Between 0 and 1, but *negative* if the model is worse than just predicting the mean
    - Easier to interpret (higher is better).

- R squared: 1 - ratio of $\sum_i (y_{pred_i} - y_{actual_i})^2$ (blue) and $\sum_i (y_{mean} - y_{actual_i})^2$ (red)



Visualizing errors

- Prediction plot (left): predicted vs actual target values
- Residual plot (right): residuals vs actual target values
    - Over- and underpredictions can be given different costs



Using evaluation metrics in model selection

- You typically want to use AUC or other relevant measures in `cross_val_score` and `GridSearchCV` instead of the default accuracy.
- scikit-learn makes this easy through the `scoring` argument
    - But, you need to need to look the mapping between the scorer and the metric (http://scikit-learn.org/stable/modules/model_evaluation.html#model-evaluation)

Scoring	Function	Comment
Classification		
‘accuracy’	`metrics.accuracy_score`	
‘average_precision’	`metrics.average_precision_score`	
‘f1’	`metrics.f1_score`	for binary targets
‘f1_micro’	`metrics.f1_score`	micro-averaged
‘f1_macro’	`metrics.f1_score`	macro-averaged
‘f1_weighted’	`metrics.f1_score`	weighted average
‘f1_samples’	`metrics.f1_score`	by multilabel sample
‘neg_log_loss’	`metrics.log_loss`	requires `predict_proba` support
‘precision’ etc.	`metrics.precision_score`	suffixes apply as with ‘f1’
‘recall’ etc.	`metrics.recall_score`	suffixes apply as with ‘f1’
‘roc_auc’	`metrics.roc_auc_score`	
Clustering		
‘adjusted_rand_score’	`metrics.adjusted_rand_score`	
Regression		
‘neg_mean_absolute_error’	`metrics.mean_absolute_error`	
‘neg_mean_squared_error’	`metrics.mean_squared_error`	
‘neg_median_absolute_error’	`metrics.median_absolute_error`	
‘r2’	`metrics.r2_score`	

Or simply look up like this:

```
Available scorers:
['accuracy', 'adjusted_mutual_info_score', 'adjusted_rand_score', 'average
_precision', 'balanced_accuracy', 'brier_score_loss', 'completeness_scor
e', 'explained_variance', 'f1', 'f1_macro', 'f1_micro', 'f1_samples', 'f1_
weighted', 'fowlkes_mallows_score', 'homogeneity_score', 'jaccard', 'jacca
rd_macro', 'jaccard_micro', 'jaccard_samples', 'jaccard_weighted', 'max_er
ror', 'mutual_info_score', 'neg_log_loss', 'neg_mean_absolute_error', 'neg
_mean_squared_error', 'neg_mean_squared_log_error', 'neg_median_absolute_e
rror', 'normalized_mutual_info_score', 'precision', 'precision_macro', 'pr
ecision_micro', 'precision_samples', 'precision_weighted', 'r2', 'recall',
'recall_macro', 'recall_micro', 'recall_samples', 'recall_weighted', 'roc_
auc', 'v_measure_score']
```

Cross-validation with AUC

```
roc_auc =  cross_val_score(SVC(), digits.data, digits.target == 9,
                           scoring="roc_auc")
print("AUC scoring: {}".format(roc_auc))
```

```
Default scoring: [0.9 0.9 0.9]
Explicit accuracy scoring: [0.9 0.9 0.9]
AUC scoring: [0.994 0.99  0.996]
```

Grid Search with accuracy and AUC

- With accuracy, gamma=0.0001 is selected
- With AUC, gamma=0.01 is selected
    - Actually has better accuracy on the test set

```
Grid-Search with accuracy
Best parameters: {'gamma': 0.0001}
Best cross-validation score (accuracy)): 0.970
Test set AUC: 0.992
Test set accuracy: 0.973

Grid-Search with AUC
Best parameters: {'gamma': 0.01}
Best cross-validation score (AUC): 0.997
Test set AUC: 1.000
Test set accuracy: 1.000
```

Final thoughts

- There exist techniques to correct label imbalance
    - Undersample the majority class, or oversample the minority class
    - SMOTE (Synthetic Minority Oversampling TEchnique) adds artificial *training* points by interpolating existing minority class points
        - Think twice before creating 'artificial' training data
- Cost-sensitive classification (not in sklearn)
    - *Cost matrix*: a confusion matrix with a costs associated to every possible type of error
    - Some algorithms allow optimizing on these costs instead of their usual loss function
    - Meta-cost: builds ensemble of models by relabeling training sets to match a given cost matrix
        - Black-box: can make any algorithm cost sensitive (but slower and less accurate)

- There are many more metrics to choose from
    - Cohen's Kappa: accuracy, taking into account the possibility of predicting the right class by chance
        - 1: perfect prediction, 0: random prediction, negative: worse than random
        - With p_0 = accuracy, and p_e = accuracy of random classifier:

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

    - Balanced accuracy: accuracy where each sample is weighted according to the inverse prevalence of its true class
        - Identical to macro-averaged recall
    - Matthews correlation coefficient: another measure that can be used on imbalanced data
        - 1: perfect prediction, 0: random prediction, -1: inverse prediction

$$MCC = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}$$

Bias-Variance decomposition

- When we repeat evaluation procedures multiple times, we can distinguish two sources of errors:
    - Bias: systematic error (independent of the training sample). The classifier always gets certain points wrong
    - Variance: error due to variability of the model with respect to the training sample. The classifier predicts some points accurately on some training sets, but inaccurately on others.
- There is also an intrinsic (noise) error, but there's nothing we can do against that.
- Bias is associated with underfitting, and variance with overfitting
- Bias-variance trade-off: you can often exchange variance for bias through regularization (and vice versa)
    - The challenge is to find the right trade-off (minimizing total error)
- Useful to understand how to tune or adapt learning algorithm

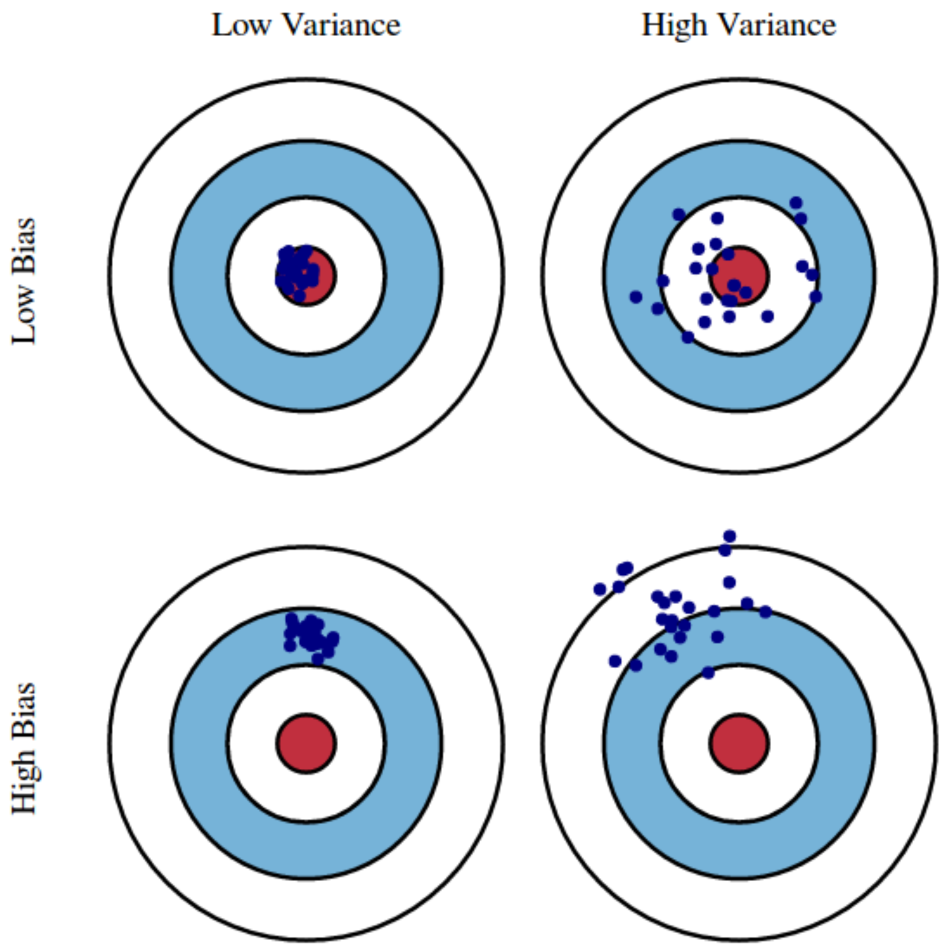


Fig. 1 Graphical illustration of bias and variance.

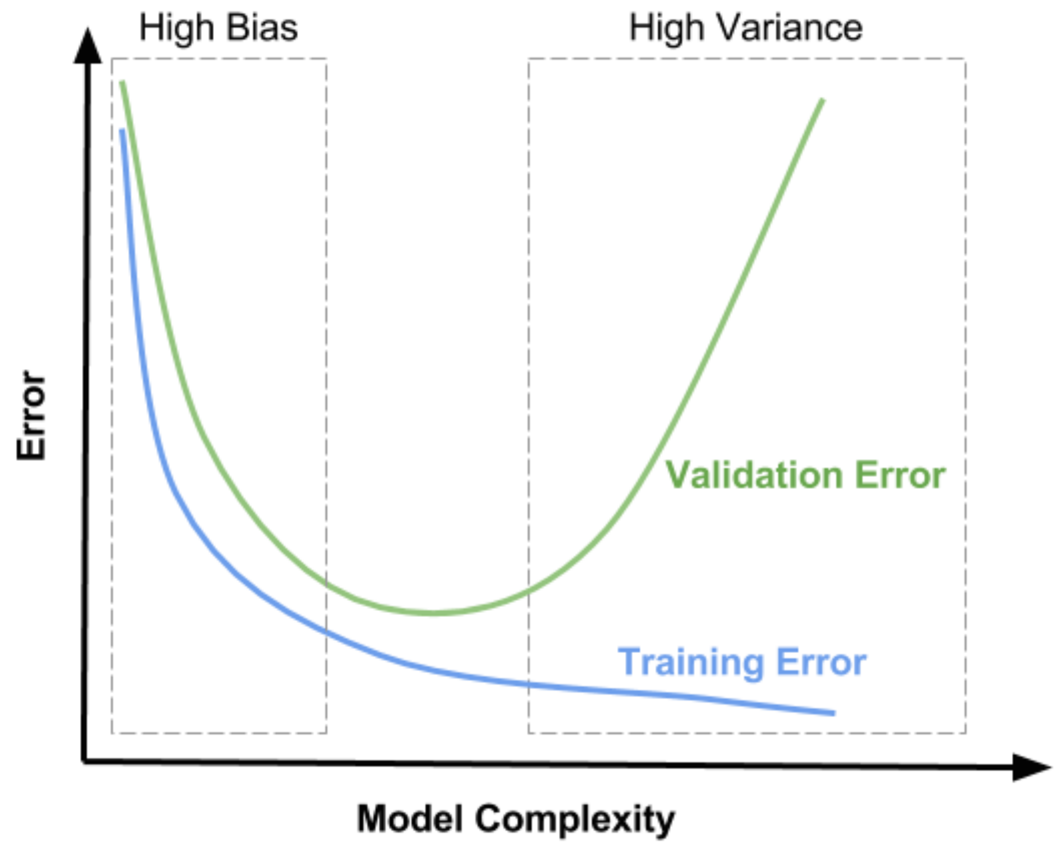
- Sadly, this is not yet supported by scikit-learn
- How to measure bias and variance (for regression):
  - Take 100 or more bootstraps (or shuffle-splits)
  - For each data point x:
    - $bias(x)^2 = (x_{true} - mean(x_{predicted}))^2$
    - $variance(x) = var(x_{predicted})$
  - Total bias: $\sum_x bias(x)^2 * w_x$, with w_x the ratio of x occuring in the test set
  - Total variance: $\sum_x variance(x) * w_x$

- General procedure for (binary) classification:
    - Take 100 or more bootstraps (or shuffle-splits)
    - Bias for any point x = misclassification ratio
        - If misclassified 50% of the time: $bias(x) = 0.5$
    - Variance for any point x is $(1 - (P(class_1)^2 + P(class_2)^2))/2$
        - $P(class_i)$ is ratio of class i predictions
        - When each class predicted half of the time: $variance(x) = (1 - (0.5^2 + 0.5^2))/2 = 0.25$
    - Total bias: $\sum_x bias(x)^2 * w_x$, with w_x the ratio of x occuring in the test data
    - Total variance: $\sum_x variance(x) * w_x$

```
for i, (train_index, test_index) in enumerate(shuffle_split.split(X)):
    clf.fit(X[train_index], y[train_index])
    y_pred = clf.predict(X[test_index])
    # Store predictions
    for i,index in enumerate(test_index):
        y_all_pred[index].append(y_pred[i])

# Compute bias, variance, error
bias_sq = sum([ (1 - x.count(y[i])/len(x))**2 * len(x)/n_repeat
            for i,x in enumerate(y_all_pred)])
var = sum([((1 - ((x.count(0)/len(x))**2 + (x.count(1)/len(x))**2))/2) *
len(x)/n_repeat
           for i,x in enumerate(y_all_pred)])
error = sum([ (1 - x.count(y[i])/len(x)) * len(x)/n_repeat
            for i,x in enumerate(y_all_pred)])
```

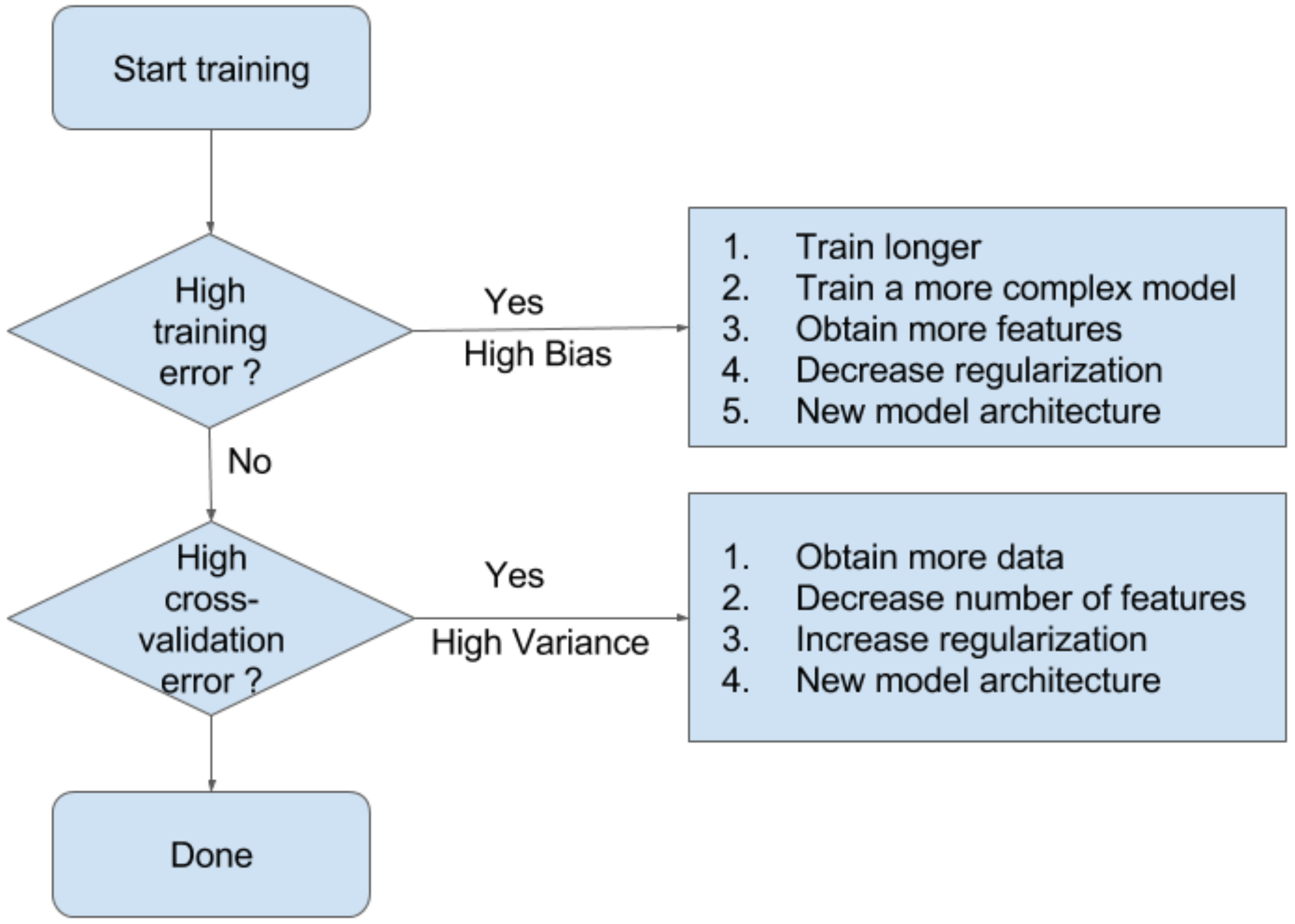
```
Bias squared: 14.50, Variance: 0.84, Total error: 15.34
```

Bias-variance and overfitting



- High bias means that you are likely underfitting
    - Do less regularization
    - Use a more flexible/complex model (another algorithm)
    - Use a bias-reduction technique (e.g. boosting, see later)
- High variance means that you are likely overfitting
    - Use more regularization
    - Get more data
    - Use a simpler model (another algorithm)
    - Use a variance-reduction techniques (e.g. bagging, see later)

Bias-Variance Flowchart (Andrew Ng, Coursera)



Start training
High training error ?
Yes
High Bias
1. Train longer
2. Train a more complex model
3. Obtain more features
4. Decrease regularization
5. New model architecture
No
High cross-validation error ?
Yes
High Variance
1. Obtain more data
2. Decrease number of features
3. Increase regularization
4. New model architecture
Done

Hyperparameter tuning

Now that we know how to evaluate models, we can improve them by tuning their hyperparameters

We can basically use any optimization technique to optimize hyperparameters:

- **Grid search**
- **Random search**

More advanced techniques:

- Local search
- Racing algorithms
- Model-based optimization (see later)
- Multi-armed bandits
- Genetic algorithms

Grid Search

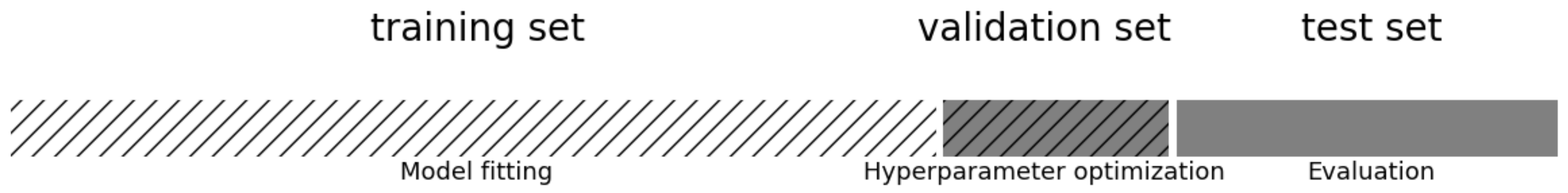
- For each hyperparameter, create a list of interesting/possible values
    - E.g. For kNN: k in [1,3,5,7,9,11,33,55,77,99]
- Evaluate all possible combination of hyperparameter values
    - E.g. using cross-validation
- Select the hyperparameter values yielding the best results
- A naive approach would be to just loop over all combinations

```python
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination, train and evaluate an SVC
        svm = SVC(gamma=gamma, C=C);
        svm.fit(X_train, y_train);
        score = svm.score(X_test, y_test)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
```

```
Size of training set: 112   size of test set: 38
Best score: 0.97
Best parameters: {'C': 100, 'gamma': 0.001}
```

Overfitting the parameters and the validation set

- Simply taking the best performing model yields optimistic results
- We've already used the test data to evaluate each hyperparameter setting!
- Hence, we don't have an independent test set to evaluate these hyperparameter settings
    - Information 'leaks' from test set into the final model
- Solution: Set aside part of the training data to evaluate the hyperparameter settings
    - Select best hyperparameters on validation set
    - Rebuild the model on the training+validation set
    - Evaluate optimal model on the test set



```
# split data into train+validation set and test set
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
# split train+validation set into training and validation set
X_train, X_valid, y_train, y_valid = train_test_split(
    X_trainval, y_trainval, random_state=1)
```

```
Size of training set: 84   size of validation set: 28   size of test set:
38

Best score on validation set: 0.96
Best parameters:  {'C': 10, 'gamma': 0.001}
Test set score with best parameters: 0.92
```

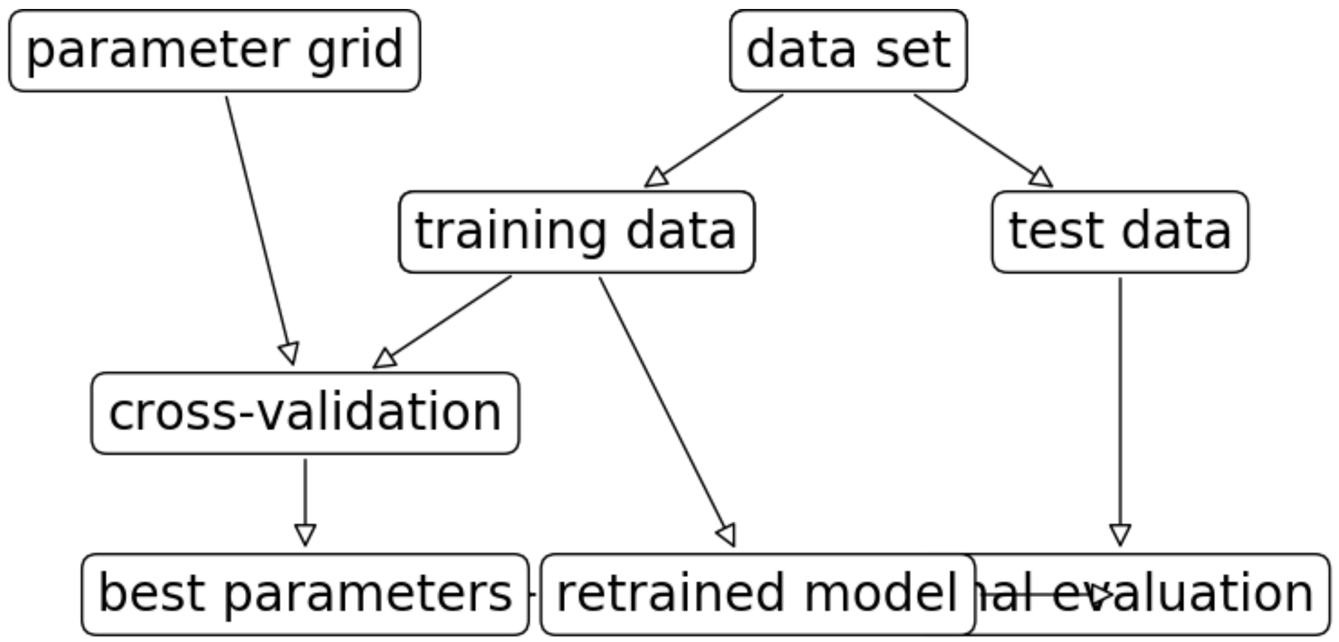
Grid-search with cross-validation

- Same problem as before: the way that we split the data into training, validation, and test set may have a large influence on estimated performance
- We need to use cross-validation again, instead of a single split
- Expensive. Often, 3 or 5-fold CV is enough

```python
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # train an SVC
        svm = SVC(gamma=gamma, C=C)
        # perform cross-validation
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # compute mean cross-validation accuracy
        score = np.mean(scores)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
```

```
Out[51]:  SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
              decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
              max_iter=-1, probability=False, random_state=None, shrinking=True,
              tol=0.001, verbose=False)
```

Overall process



Grid search in scikit-learn

- Create a parameter grid as a dictionary
    - Keys are parameter names
    - Values are lists of hyperparameter values

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
print("Parameter grid:\n{}".format(param_grid))
```

```
Parameter grid:
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 1
00]}
```

- `GridSearchCV`: like a classifier that uses CV to automatically optimize its hyperparameters internally
    - Input: (untrained) model, parameter grid, CV procedure
    - Output: optimized model on given training data
    - Should only have access to training data

```
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

Out[56]:

```
GridSearchCV(cv=5, error_score='raise-deprecating',
             estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef
0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='auto_deprecated', kernel='rbf', max_ite
r=-1,
                           probability=False, random_state=None, shrinkin
g=True,
                           tol=0.001, verbose=False),
             iid='warn', n_jobs=None,
             param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100],
                         'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=Fals
e,
             scoring=None, verbose=0)
```

The optimized test score and hyperparameters can easily be retrieved:

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

```
Test set score: 0.97
```

```
grid_search.best_params_
grid_search.best_score_
```

```
Best parameters: {'C': 100, 'gamma': 0.01}
Best cross-validation score: 0.97
```

```
grid_search.best_estimator_
```

```
Best estimator:
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

Visualizing hyperparameter impact

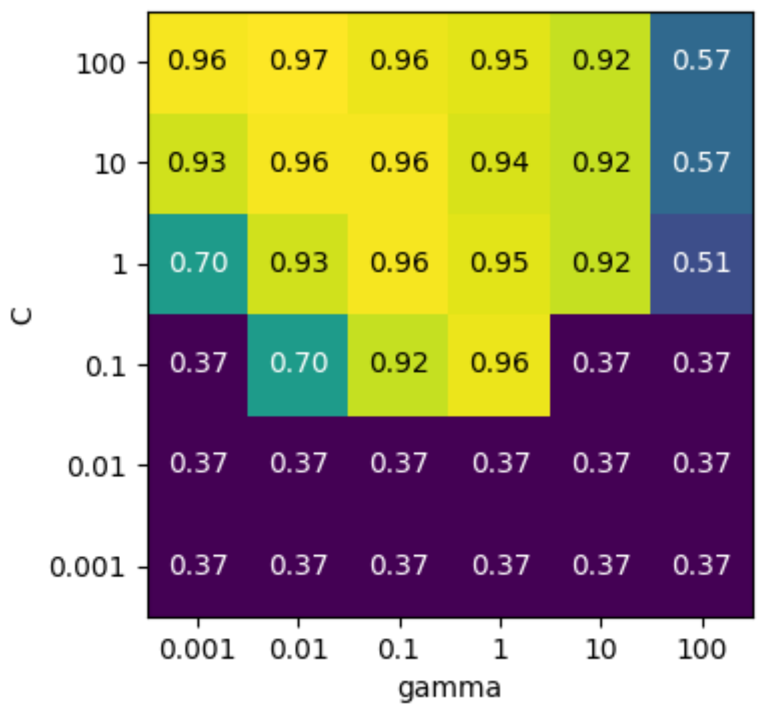
We can retrieve and visualize the cross-validation resulst to better understand the impact of hyperparameters

```
results = pd.DataFrame(grid_search.cv_results_)
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	...	split4_test_score	mean_test_score	std_test_score	rank_test_score
0	1.06e-03	5.41e-04	4.11e-04	1.21e-04	...	0.38	0.37	0.01	22
1	5.87e-04	5.28e-05	2.51e-04	7.88e-06	...	0.38	0.37	0.01	22
2	6.44e-04	8.58e-05	3.07e-04	1.02e-04	...	0.38	0.37	0.01	22
3	8.14e-04	1.50e-04	3.85e-04	6.89e-05	...	0.38	0.37	0.01	22
4	9.86e-04	2.02e-04	4.03e-04	1.13e-04	...	0.38	0.37	0.01	22

5 rows × 15 columns

Visualize as a heatmap



When hyperparameters depend on other parameters, we can use lists of dictionaries to define the hyperparameter space

```
param_grid = [{'kernel': ['rbf'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100],
               'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
              {'kernel': ['linear'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
```

```
List of grids:
[{'kernel': ['rbf'], 'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001,
0.01, 0.1, 1, 10, 100]}, {'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1,
10, 100]}]
```

Nested cross-validation

- Note that we are still using a single split to create the outer test set
- We can also use cross-validation here
- Nested cross-validation:
    - Outer loop: split data in training and test sets
    - Inner loop: run grid search, splitting the training data into train and validation sets
- Result is a just a list of scores
    - There will be multiple optimized models and hyperparameter settings (not returned)
- To apply on future data, we need to train `GridSearchCV` on all data again

```
scores = cross_val_score(GridSearchCV(SVC(), param_grid, cv=5),
                         iris.data, iris.target, cv=5)
```

```
Cross-validation scores:  [0.967 1.    0.9   0.967 1.   ]
Mean cross-validation score:  0.9666666666666668
```

Parallelizing cross-validation and grid-search

- On a practical note, it is easy to parallellize CV and grid search
- `cross_val_score` and `GridSearchCV` have a `n_jobs` parameter defining the number of cores it can use.
    - set it to `n_jobs=-1` to use all available cores.

Random Search

- Grid Search has a few downsides:
    - Optimizing many hyperparameters creates a combinatorial explosion
    - You have to predefine a grid, hence you may jump over optimal values
- Random Search:
    - Picks `n_iter` random parameter values
    - Scales better, you control the number of iterations
    - Often works better in practice, too
        - not all hyperparameters interact strongly
        - you don't need to explore all combinations

- Executing random search in scikit-learn:
  - `RandomizedSearchCV` works like `GridSearchCV`
  - Has `n_iter` parameter for the number of iterations
  - Search grid can use distributions instead of fixed lists

```
param_grid = {'C': expon(scale=100),
              'gamma': expon(scale=.1)}
random_search = RandomizedSearchCV(SVC(), param_distributions=param_grid
,
                                   n_iter=20)
random_search.fit(X_train, y_train)
```

```
Out[64]: RandomizedSearchCV(cv='warn', error_score='raise-deprecating',
                   estimator=SVC(C=1.0, cache_size=200, class_weight=Non
e,
                                 coef0=0.0, decision_function_shape='ov
r',
                                 degree=3, gamma='auto_deprecated',
                                 kernel='rbf', max_iter=-1, probability=F
alse,
                                 random_state=None, shrinking=True, tol=
0.001,
                                 verbose=False),
                   iid='warn', n_iter=20, n_jobs=None,
                   param_distributions={'C': <scipy.stats._distn_infrastr
ucture.rv_frozen object at 0x11d861668>,
                                        'gamma': <scipy.stats._distn_infr
astructure.rv_frozen object at 0x11d98a860>},
                   pre_dispatch='2*n_jobs', random_state=None, refit=Tru
e,
                   return_train_score=False, scoring=None, verbose=0)
```

Summary

- k-fold Cross-validation
    - Choose k depending on how much data you have
        - Larger k is slower, but allows more training data
        - 10-fold, 5-fold, 5x2-fold most popular
    - Always use stratification for (imbalanced) classification
    - Train-test split and Shuffle-split: useful for large datasets
    - Use grouping when you want to generalize over groups
- Model selection
    - Don't aggregate over test scores: those have seen the test data
    - Use validation sets to choose algorithms/hyperparameters first
- Optimization
    - Grid Search: exhaustive but simple
    - Random Search: scales better
    - We'll see more advanced techniques later

Summary

- Real-world data is often imbalanced
- False positives may be much worse than false negatives (or vise-versa)
- Binary classification
  - Select metrics that can distinguish different types of errors (precision, recall, f1-score, AUC,...)
  - Calibrate decision thresholds to the task at hand
  - Precision-Recall and ROC curves: choose the best threshold or take area under the curve
- Multiclass classification
  - Macro/Micro/weighted average of per-class scores (one-vs-all)
- Regression
  - (Root) mean squared/absolute error from 0..Inf
  - R2 easier to interpret
- All measures can be used in cross-validation or grid/random search
- Cost-sensitive classification: optimize for any cost matrix or cost function