Model Selection

Evaluating and selecting algorithms and hyperparameters.

Evaluating models

To know whether we can *trust* what our algorithm has learned, we need to evaluate it.

We will focus on supervised methods (classification and regression)

- With the labels we can objectively evaluate models
- There are still many ways to do this

In unsupervised learning (e.g. clustering) we don't know what the 'right' output should be

- We can only optimize for certain properties (e.g. cluster purity)
- Ultimately, we need to inspect results manually (or create an external evaluation measure)

The holdout (simple train-test split)

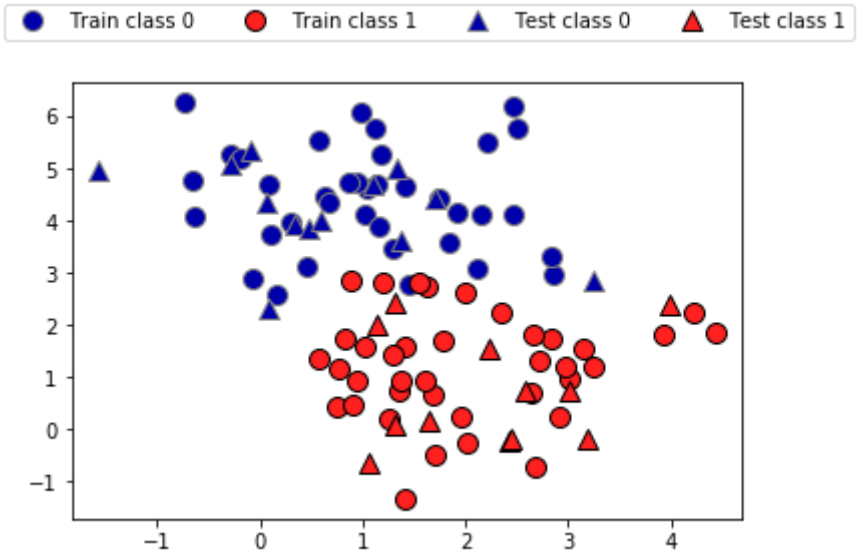
We've already seen the most basic form of evaluation:

- Split data into training and test set (75%-25%)
  - In sklearn we split in training and test predictors (X_train, X_test) and labels (y_train, y_test)
- 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

```
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
)
# Instantiate a model and fit it to the training set
model = LogisticRegression().fit(X_train, y_train)
# evaluate the model on the test set
print("Test set score: {:.2f}".format(model.score(X_test, y_test)))
```

```
Test set score: 0.92
```

Visualized

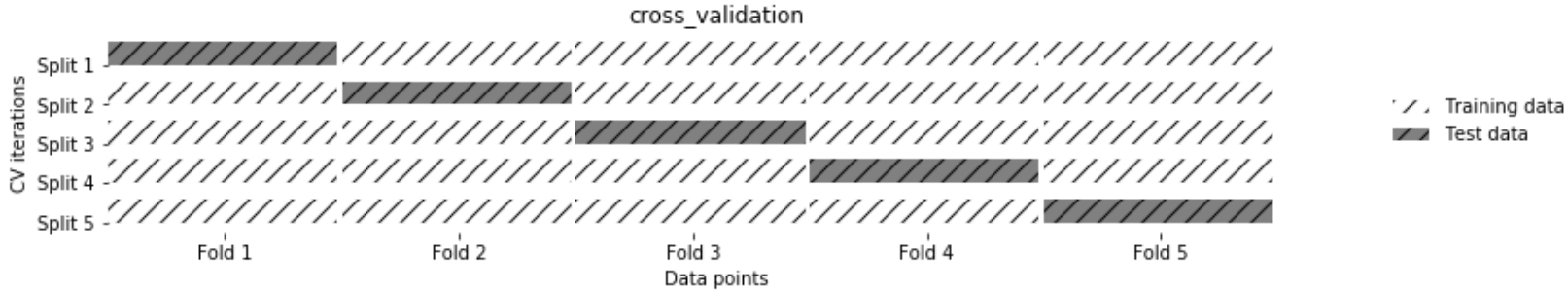

Train class 0
Train class 1
Test class 0
Test class 1
6
5
4
3
2
1
0
−1
−1
0
1
2
3
4

Limitations to this approach:

- Why 75%? Are there better ways to split?
- What if one random split yields different models (and scores) than another?
- What if all examples of one class all end up in the training/test set?

Cross-validation

- More stable, thorough way to estimate generalization performance
- *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)



Cross-validation in scikit-learn

- `cross_val_score` function with learner, training data, labels
- Returns list of all scores
  - Does 3-fold CV by default
  - Default scoring measures are accuracy (classification) or R^2 (regression)
- Even though models are built internally, they are not returned

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

```
Cross-validation scores: [ 0.961  0.922  0.958]
```

- Change the number of folds with `cv` parameter
- 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

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

```
Cross validation scores: [ 1.     0.967  0.933  0.9    1.   ]
```

- Aggregate the scores yourself (e.g. mean)
- This means that the model is 96% accurate *on average*

```
print("Average cross-validation score: {:.2f}".format(scores.mean()))
print("Variance in cross-validation score: {:.4f}".format(np.var(scores
)))
```

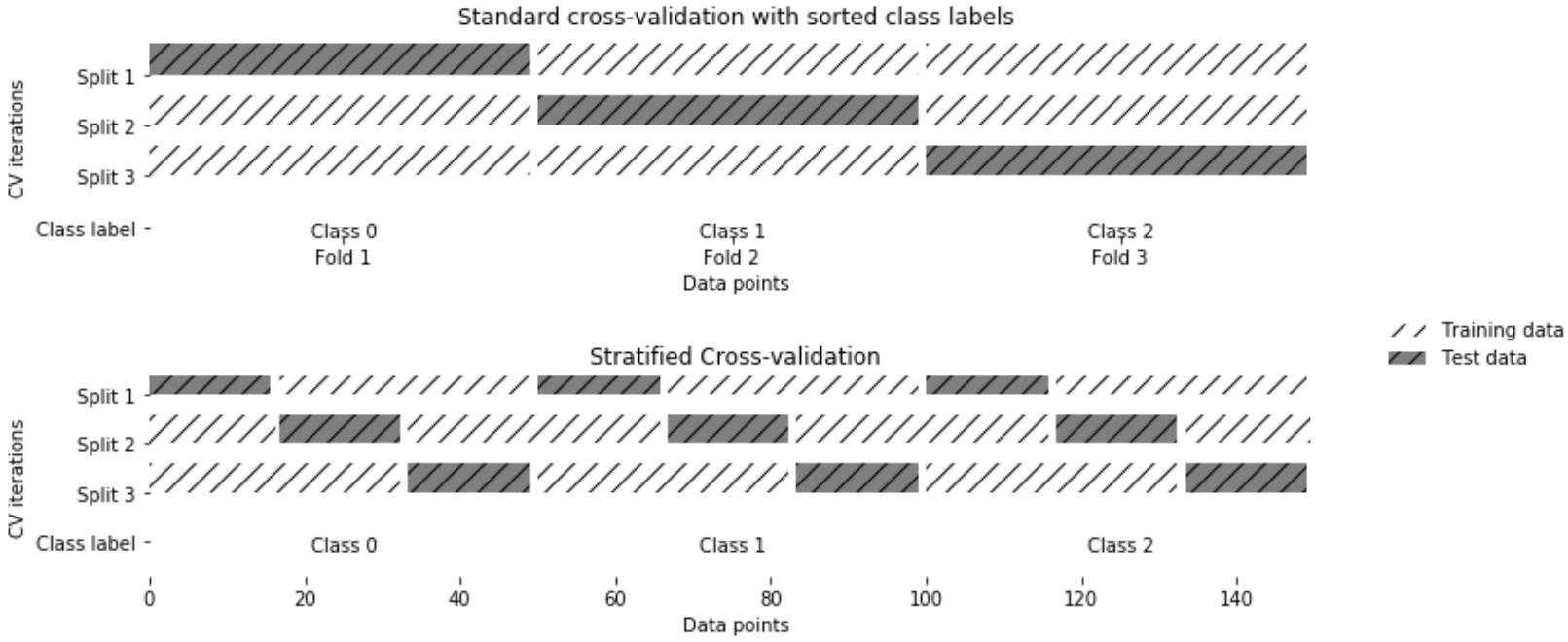
```
Average cross-validation score: 0.96
Variance in cross-validation score: 0.0015
```

Benefits 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

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



- scikit-learn (e.g. `cross_val_score`) will use:
  - 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`
  - randomizable (`shuffle` parameter), non-stratified (!)

```
kfold = KFold(n_splits=5)
cross_val_score(logreg, iris.data, iris.target, cv=kfold
```

```
Cross-validation scores KFold(n_splits=5):
[ 1.     0.933  0.433  0.967  0.433]
```

- Use `StratifiedKFold` to create stratified splits
  - randomizable (`shuffle` parameter), stratified

```
kfold = StratifiedKFold(n_splits=5, shuffle=True)
cross_val_score(logreg, iris.data, iris.target, cv=kfold)
```

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

Can you explain this result?

```
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.]

Cross-validation scores (shuffled, not stratified):
[ 0.9   0.96  0.96]
Cross-validation scores (shuffled, stratified):
[ 0.941  0.941  1.   ]
Cross-validation scores (default: stratified, not shuffled):
[ 0.961  0.922  0.958]
```

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

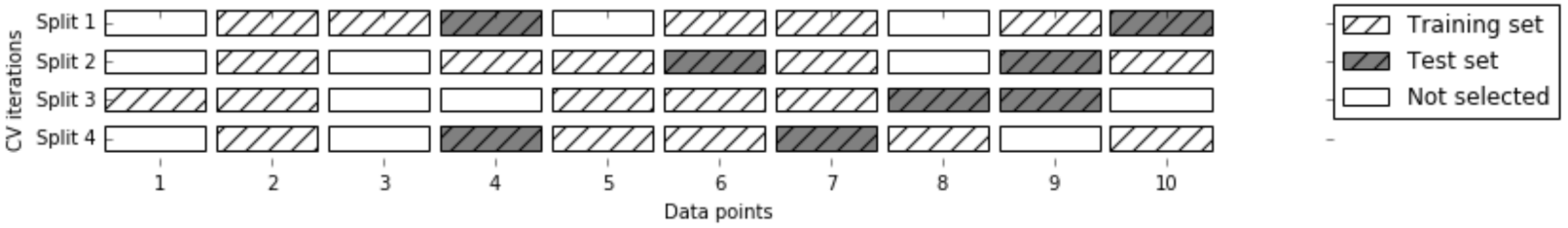
```
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
```

```
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
- Typically, the whole dataset is used each iteration (except for large datasets)

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



- In scikit-learn, `train_size` and `test_size` can be absolute numbers or a percentage of the total dataset
- Stratified variant: `StratifiedShuffleSplit' (recommended for classification)

```
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:
[ 0.92   0.973  0.92   0.947  0.92   0.893  0.933  0.96   0.947  0.933]
```

Note: this is related to *bootstrapping*:

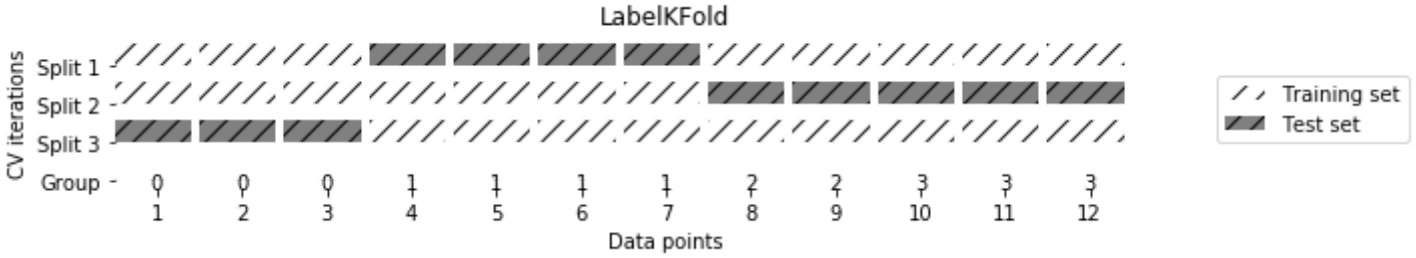
- Sample n (total number of samples) data points, with replacement, as training set (the bootstrap)
- Use the unsampled (out-of-bootstrap) samples as the test set
- Repeat `n_iter` times, obtaining `n_iter` scores
- Not supported in scikit-learn, use Shuffle-Split instead.
  - With `train_size=0.66`, `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, this is supported by `GroupKFold`
  - 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

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

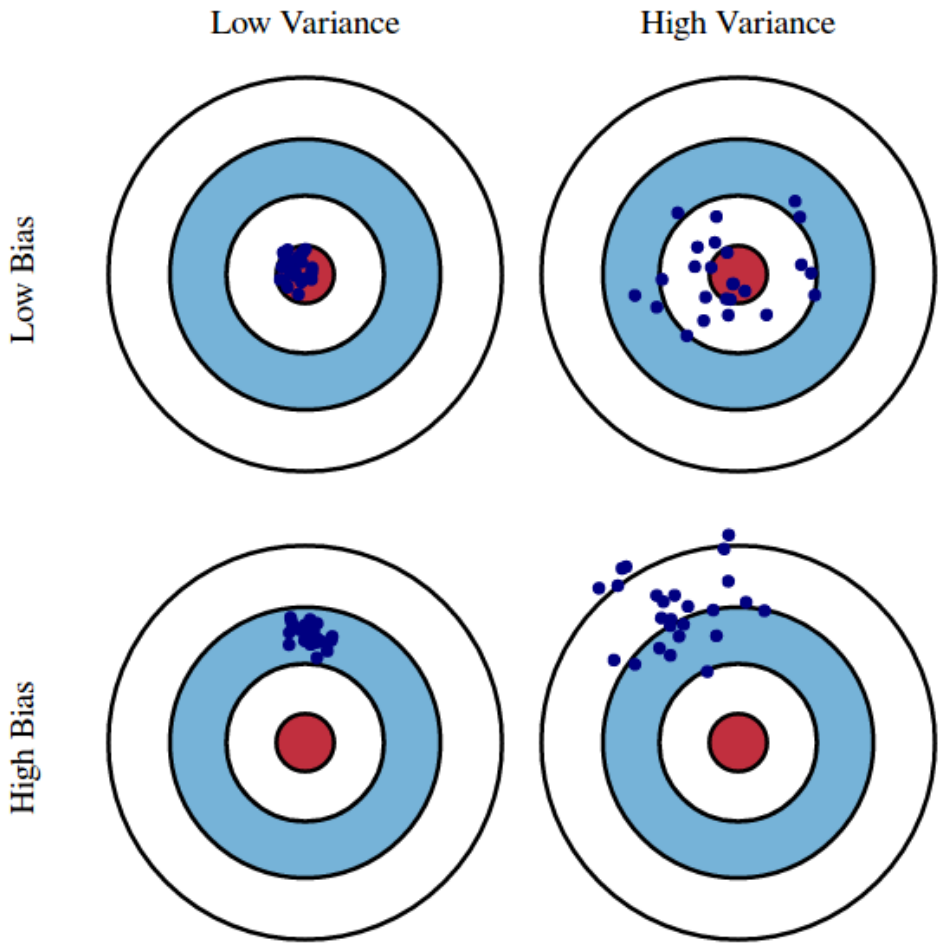


Fig. 1 Graphical illustration of bias and variance.

- 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: $\sum_x variance(x) * w_x$

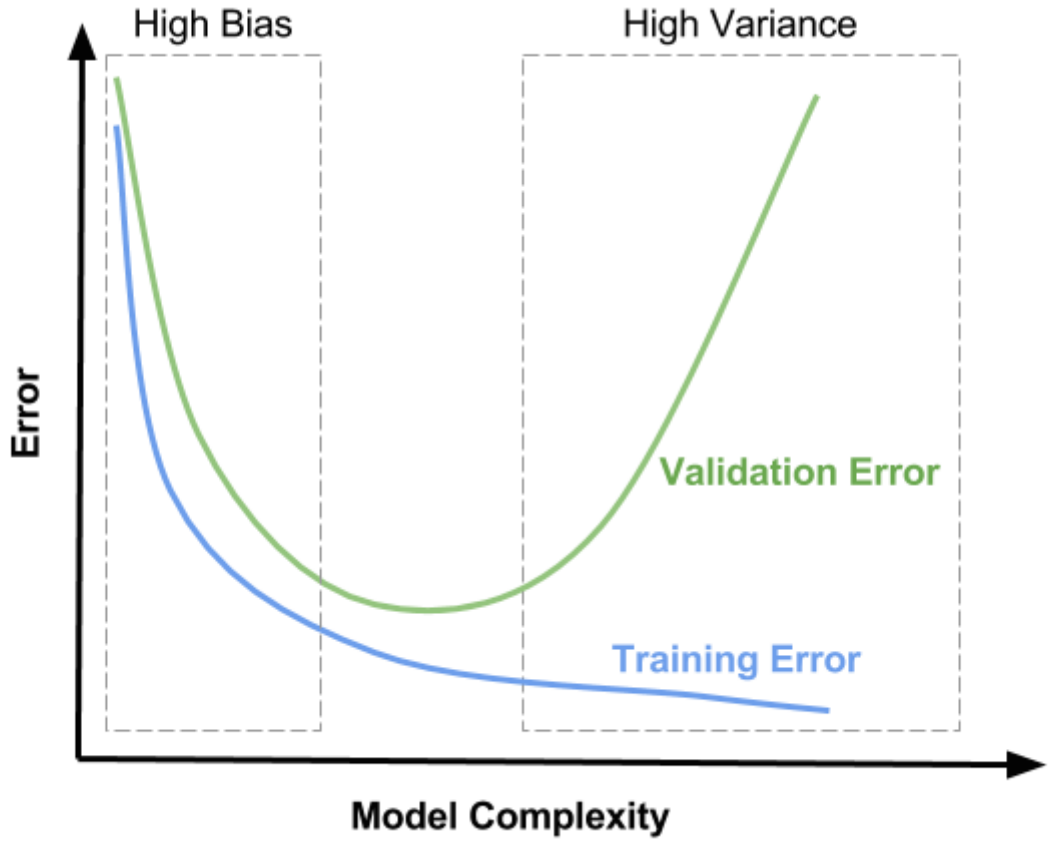
- General procedure for (binary) classification:
  - Take 100 or more bootstraps (or shuffle-splits)
  - Bias for any point x = misclassification ratio
    - If misclassified 50% of the time: $bias(x) = 0.5$
  - Variance for any point x is $(1 - (P(class_1)^2 + P(class_2)^2))/2$
    - $P(class_i)$ is 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)])
```

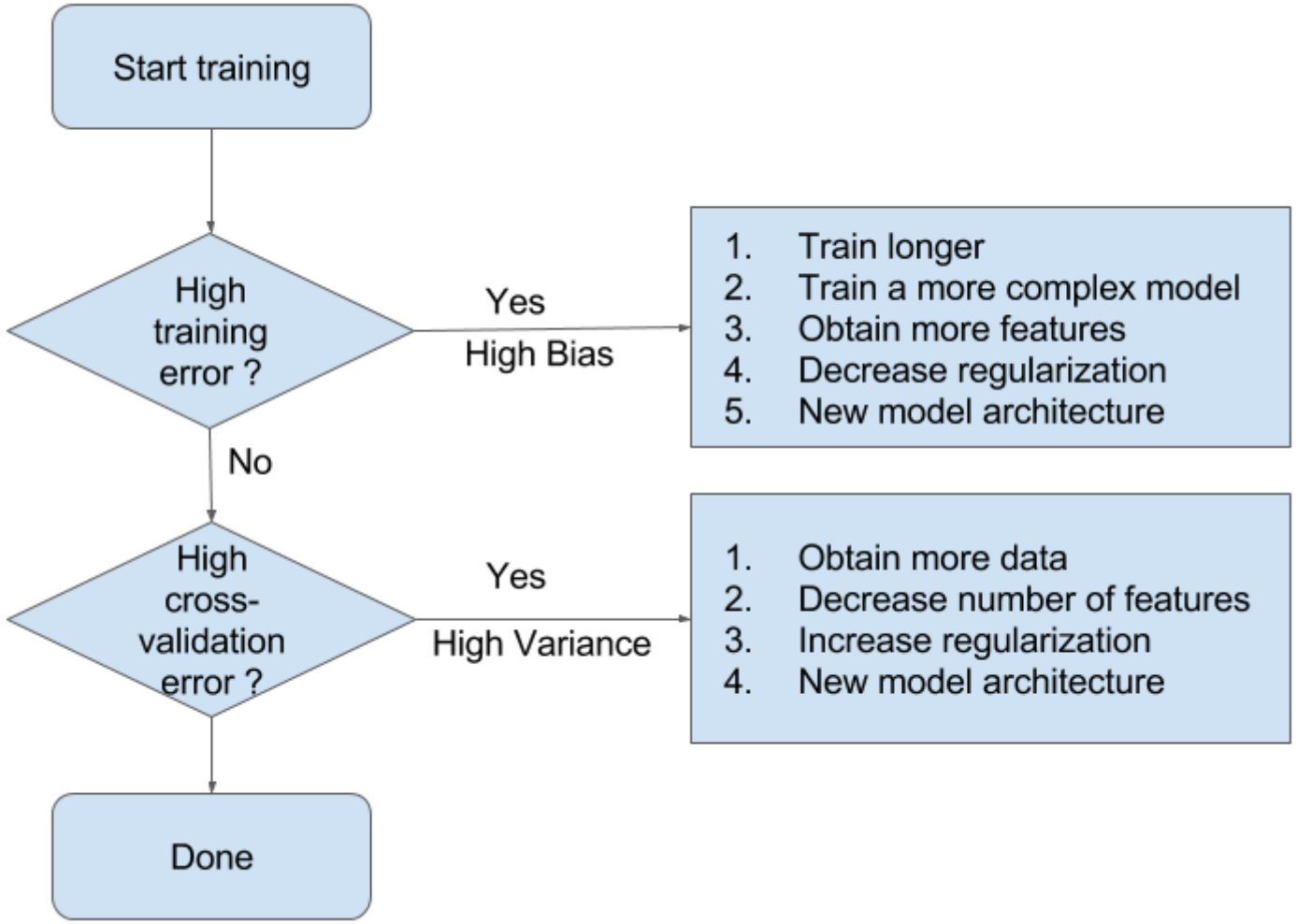
```
Bias squared: 14.23, Variance: 0.94, Total error: 15.17
```

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)



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

```
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
```

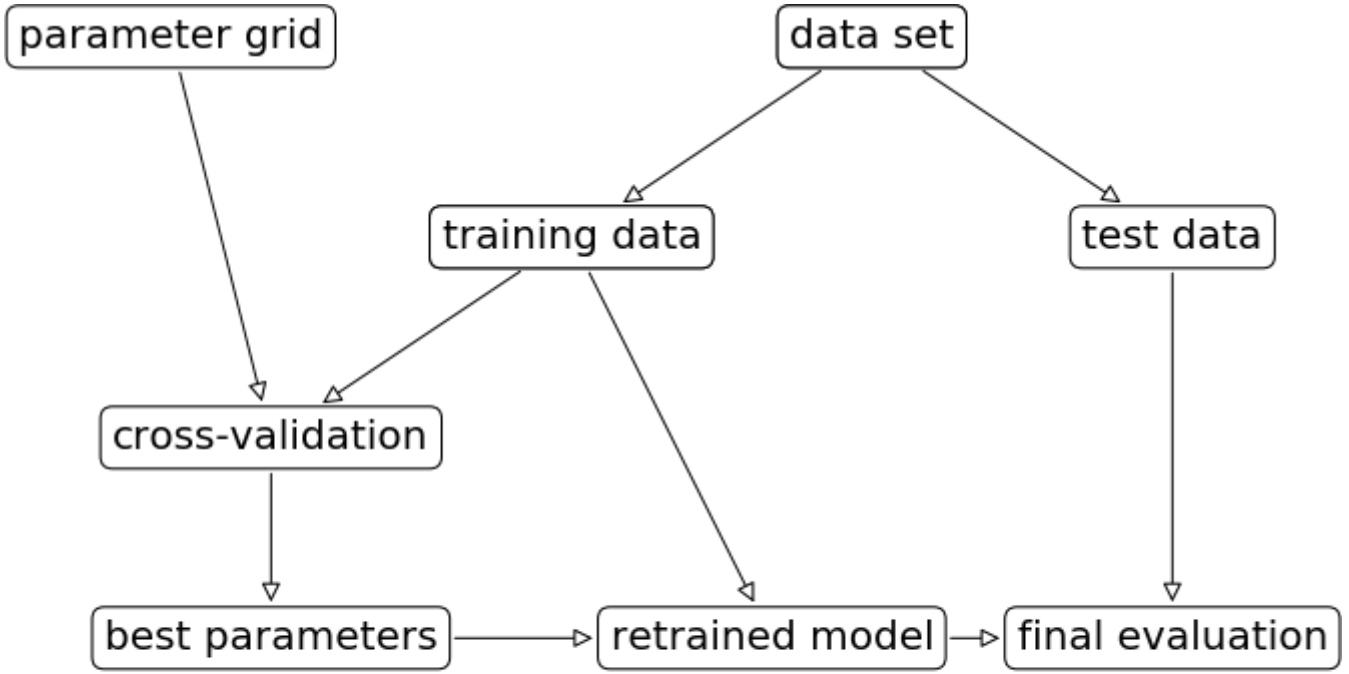
Grid-search with cross-validation

- 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

```
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[230]:
```
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[235]: GridSearchCV(cv=5, error_score='raise',
          estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
        decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
        max_iter=-1, probability=False, random_state=None, shrinking=True,
        tol=0.001, verbose=False),
          fit_params=None, iid=True, n_jobs=1,
          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='warn',
          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

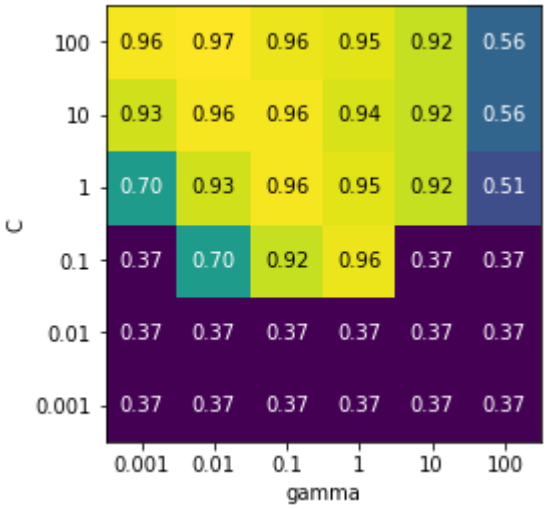
We can retrieve and visualize the cross-validation resulst to better understand the impact of hyperparameters

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

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	...
0	6.72e-04	2.71e-04	0.37	0.37	...
1	5.47e-04	2.17e-04	0.37	0.37	...
2	5.57e-04	2.39e-04	0.37	0.37	...
3	7.11e-04	2.87e-04	0.37	0.37	...
4	9.89e-04	4.13e-04	0.37	0.37	...

5 rows × 22 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:  0.966666666667
```

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[243]: RandomizedSearchCV(cv=None, error_score='raise',
          estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=
0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False),
          fit_params=None, iid=True, n_iter=20, n_jobs=1,
          param_distributions={'C': <scipy.stats._distn_infrastructure.rv
_frozen object at 0x1c21b5aac8>, 'gamma': <scipy.stats._distn_infrastruct
ure.rv_frozen object at 0x1c21b5ac18>},
          pre_dispatch='2*n_jobs', random_state=None, refit=True,
          return_train_score='warn', 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