Lab 9

Ensembles

Introduction

- Objectives
  - Learn to train ensembles of classifiers
  - Voting: hard, soft vote
  - Bagging, random forests, extremely randomized trees
  - Boosting, AdaBoost, Gradient boosting, XGBoost

Ensemble learning

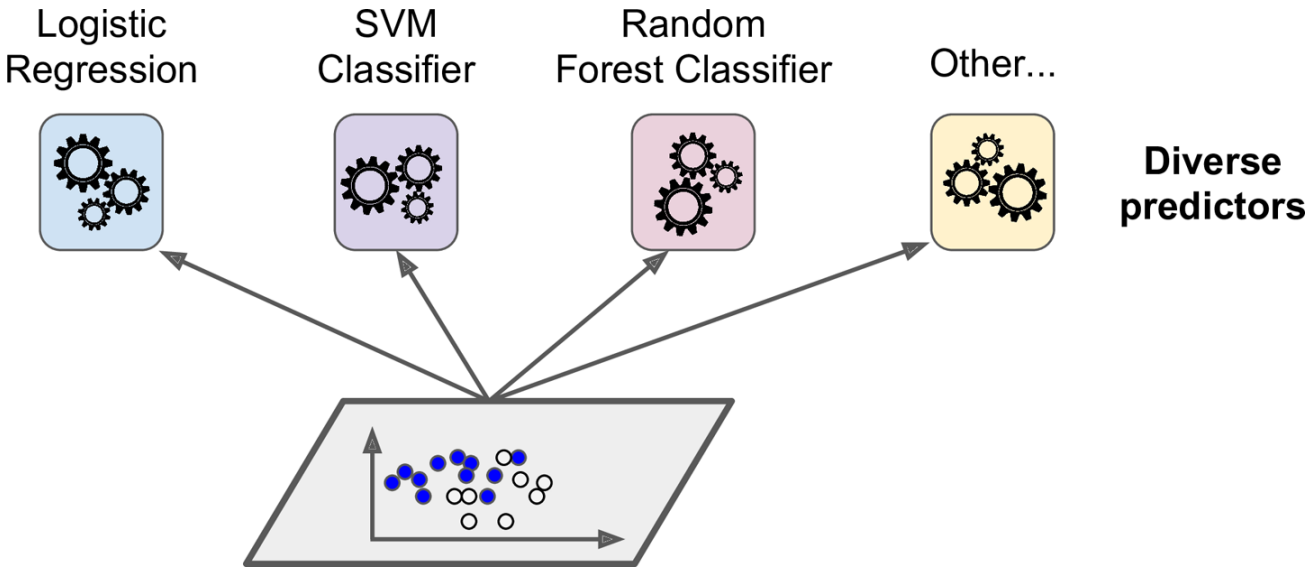
- Why does it work?
  - Different models may be good at different 'parts' of data (even if they are underfit)
  - Individual mistakes can be 'averaged' (especially if models overfit)

- Which models should be combined? According to bias-variance analysis:
  - If model underfits (high bias, low variance): combine with other low-variance models
    - Need to be different 'experts' on different parts of the data
    - Bias reduction. Can be done with **Boosting**
  - If model overfits (low bias, high variance): combine with other low-bias models
    - Need to be different: individual mistakes must be different
    - Variance reduction. Can be done with **Bagging**
  - Models must be uncorrelated but good enough (otherwise the ensemble is worse)
  - We can also learn how to combine the classifiers (**Stacking**)

Ensemble learning

Voting classifiers:

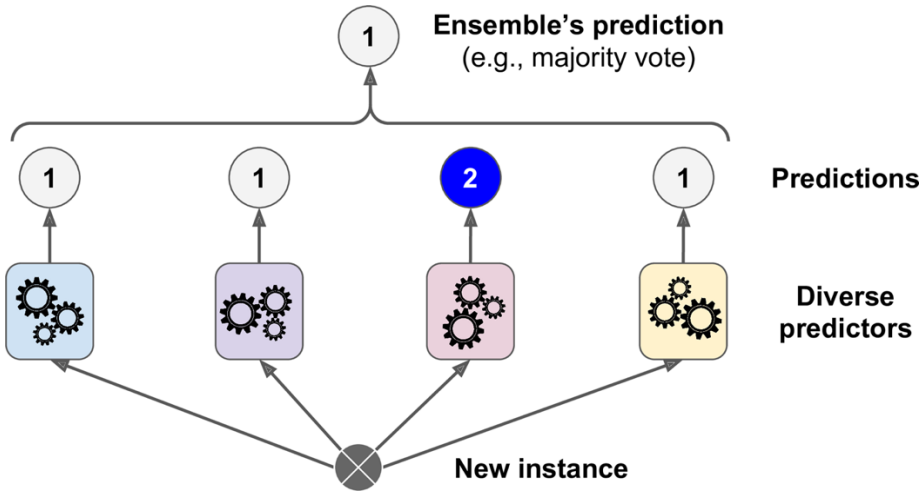
- Train diverse models and aggregate the predictions
  - **Hard vote:** majority class wins (class order breaks ties)
  - **Soft vote:** sum class probabilities p_{mc} over M models, $argmax \sum_{m=1}^{M} w_c p_{mc}$. , classes can get different weights w_c (default $w_c = 1$)



Ensemble learning

Voting classifiers:

Ex hard majority voting

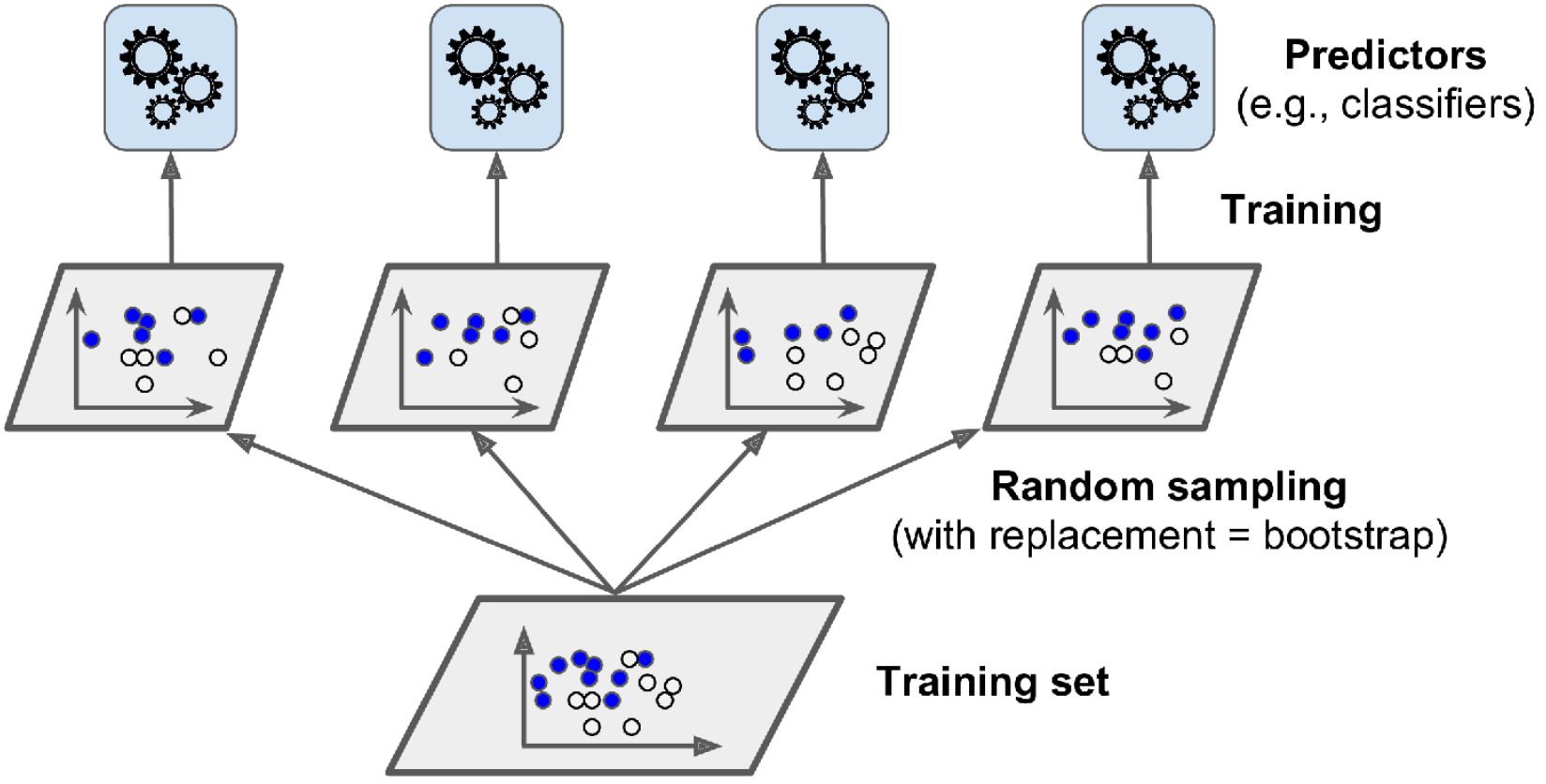



```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(solver="lbfgs", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
svm_clf = SVC(gamma="scale", random_state=42)

voting_clf = VotingClassifier(
estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
voting='hard')
```

Bagging (bootstrap aggregating)

- Obtain different models by training the same model on different training samples
  - Reduce overfitting by averaging out individual predictions (variance reduction)
- In practice: take N bootstrap samples of data, train a model on each bootstrap
  - Higher N: more models, more smoothing (but slower training and prediction)
- Base models should be unstable: different training samples yield different models
  - Eg. deep decision trees or randomized decision trees
- Prediction by averaging predictions of base models
  - Soft voting for classification (possible weighted)
- Can produce uncertainty estimates as well
  - By combining class probabilities of individual models

Bagging (bootstrap aggregating)



Random forests

- Uses **randomized trees** to make models even less correlated (more unstable)
  - At every split, only consider *max_features* features, randomly selected
  - **Extremely randomized trees**: considers only 1 random threshold for random set of features (faster to train)

- Effect on bias and variance
  - Increasing the number of models (trees) decreases variance (less overfitting)
  - Bias is mostly unaffected, but will increase if forest becomes too large (oversmoothing)

- Feature importance: computed by considering how much the tree nodes that use that feature reduce impurity on average (across all trees in the forest).

Bagging in practice

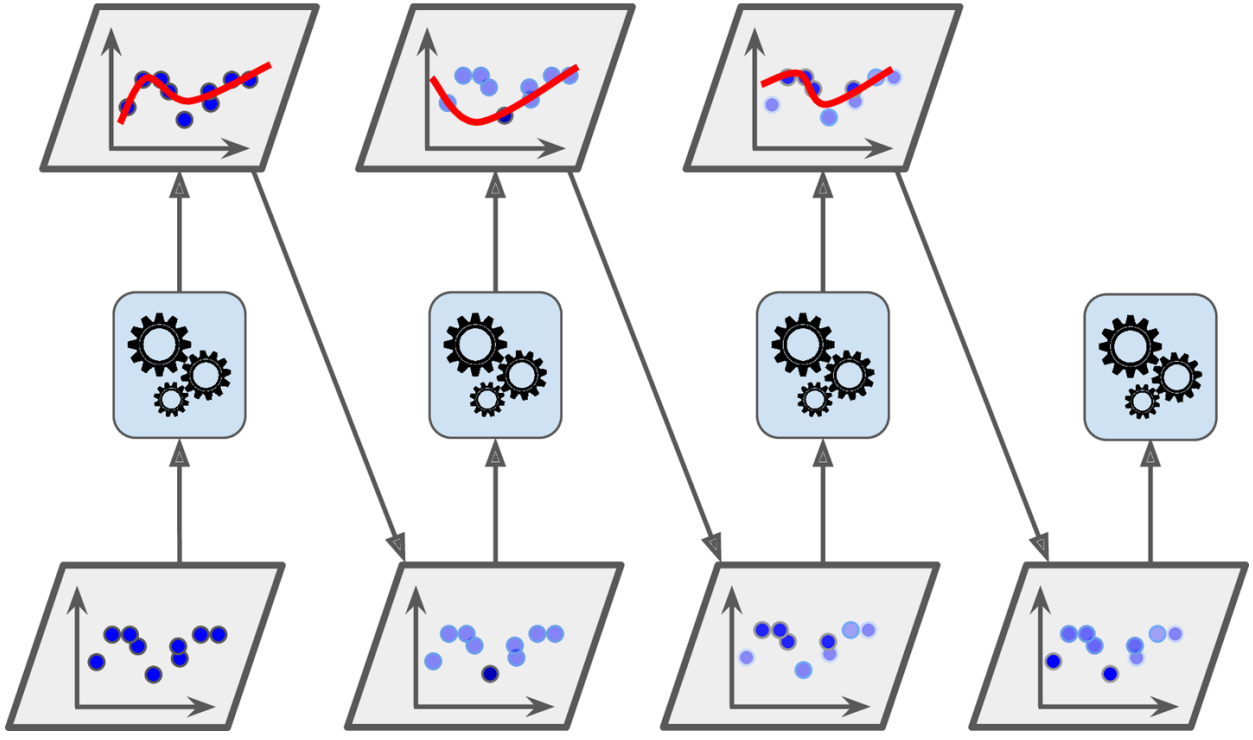
- Different implementations can be used. In scikit-learn
  - `BaggingClassifier`: you can choose base model and sampling procedure
  - `RandomForestClassifier:` default implementation, many options
  - `ExtraTreesClassifier:` uses extremely randomized trees

- Most important parameters:
  - `n_estimators` (>100, higher is better, but diminishing returns)
    - Will start to underfit (bias error component increases slightly)
  - `max_features`:
    - Defaults $sqrt(p)$ for classification
    - Set smaller to reduce space/time requirements
  - Parameters of trees: `max_depth`, `min_samples_split`
    - Prepruning useful to reduce model size, but not too much

- Easy to parallelize (set `n_jobs` to -1)
- Fix `random_state` (bootstrap samples) for reproducibility

Boosting

- Combine several weak learners into a strong learner: train predictors sequentially, each trying to correct the previous one.
- Many boosting methods available, the most popular is Adaboost



Adaptive Boosting (AdaBoost)

- Obtain different models by reweighting the training data every iteration
  - Reduce overfitting by focusing on the 'hard' training examples
- Increase weights of instances misclassified by the ensemble
- Base model should be simple so that different instance weights lead to different models
  - Underfitting models: decision stumps (or very shallow trees)
  - Each is an 'expert' on some parts of the data
- **Additive model**: predictions at iteration I are sum of base model predictions (weight by accuracy)

$$f_I(\mathbf{x}) = \sum_{i=1}^{I} w_i g_i(\mathbf{x})$$

- **Effect on bias and variance**
  - Adaboost reduces bias (and a little variance)
    - Boosting is a bias reduction technique
  - Boosting too much will eventually increase variance

Adaboost

- Example AdaBoost with ‘stumps’ (decision trees of depth=1)

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
DecisionTreeClassifier(max_depth=1), n_estimators=200,
algorithm="SAMME.R", learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)

y_pred_ada = ada_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_ada))
cmat = confusion_matrix(y_test,y_pred_ada)
print(cmat)
```

Gradient boosting

- Generalization of boosting to arbitrary differentiable loss functions
- Ensemble of models, each fixing the remaining mistakes of the previous ones
  - Each iteration, instead of changing the weights, the task is to predict the residual error of the ensemble
  - Additive model: predictions at iteration I are sum of base model predictions. Each new predictor is fitted to minimize a loss function, given the previous ensemble.
    - Learning rate (or shrinkage) η: small updates work better (reduces variance)

$$f_I(\mathbf{x}) = g_0(\mathbf{x}) + \sum_{i=1}^{I} \eta \cdot g_i(\mathbf{x}) = f_{I-1}(\mathbf{x}) + \eta \cdot g_I(\mathbf{x})$$

- **Effect on bias and variance**
  - Gradient boosting is very effective at reducing bias error
  - Boosting too much will eventually increase variance

Gradient boosting

- Among the most powerful and widely used models
- Work well on heterogeneous features and different scales
- Typically better than random forests, but requires more tuning, longer training
- Does not work well on high-dimensional sparse data

- Many hyperparameters
  - `n_estimators`: higher is better, but will start to overfit
  - `learning_rate`: lower rates mean more trees are needed to get more complex models
    - Set `n_estimators` as high as possible and tune learning_rate
    - Or choose a `learning_rate` and use early stopping to avoid overfitting
  - `max_depth`: typically kept low (<5), reduce when overfitting
  - `max_features`: can also be tuned, similar to random forests
  - `n_iter_no_change`: early stopping, algorithm stops if improvement is less than a certain tolerance `tol` for more than `n_iter_no_change` iterations
  - `loss`: loss function (default 'log_loss' cross-entropy)

Extreme Gradient Boosting (XGBoost)

- Faster version of gradient boosting: allows more iterations on larger datasets

- XGBoost in practice:
  - Not part of scikit-learn, but `HistGradientBoostingClassifier` is similar
  - The **xgboost** python package is scikit-learn compatible

    https://xgboost.readthedocs.io/en/stable/

Lab9

- Toy example
  - Voting
    - Hard
    - Soft
  - Bagging with decision trees
  - Boosting: adaboost with decision stumps
  - Gradient boosting

- MNIST with ensembles
- Optional: MNIST with XGBoost