MMA/MMAI 869
Machine Learning and AI

Ensembles

Stephen Thomas

Updated: Nov 3, 2022

Smith School of Business | Queen's University

Outline

- Ensemble methods
  - Voting/Committee
  - Bagging
  - Boosting
- Comparison

ENSEMBLES

Combining Classifiers

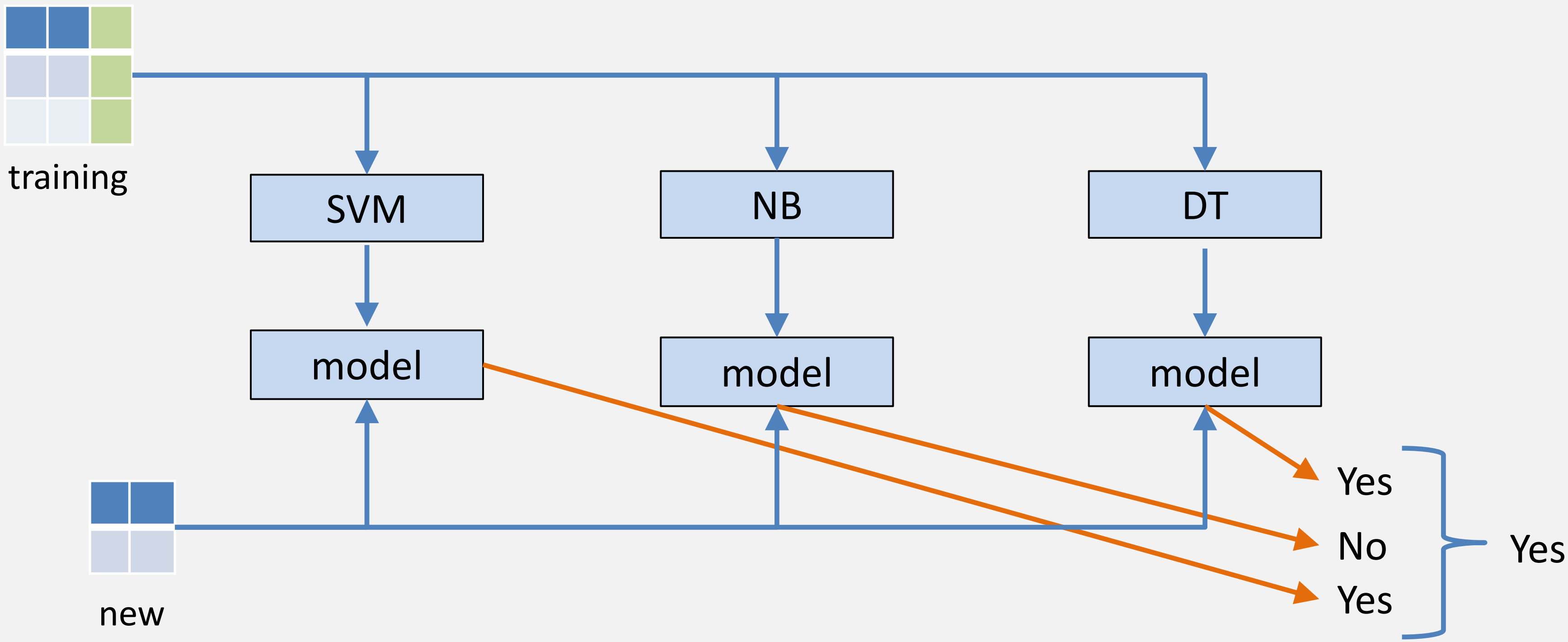
- So far, we have only discussed individual models
- **Can we combine multiple models to produce a better model?**
  – Yes! Called "ensembles" or "combinations"
- In practice, ensembles are ***very*** effective
  – E.g., ensemble of DTs shown to be better than NNs for tabular data
- Many popular ways:
  – **Committee**, *aka* **Voting**
  – **Bagging** (incl. Random Forests, Extra Trees)
  – **Boosting** (incl. Adaboost, GBM, XGBoost)
- While you can create an ensemble manually, you mostly just use one of the above

COMMITTEES

Committee

Committee is a parallel ensemble: each model is built independently

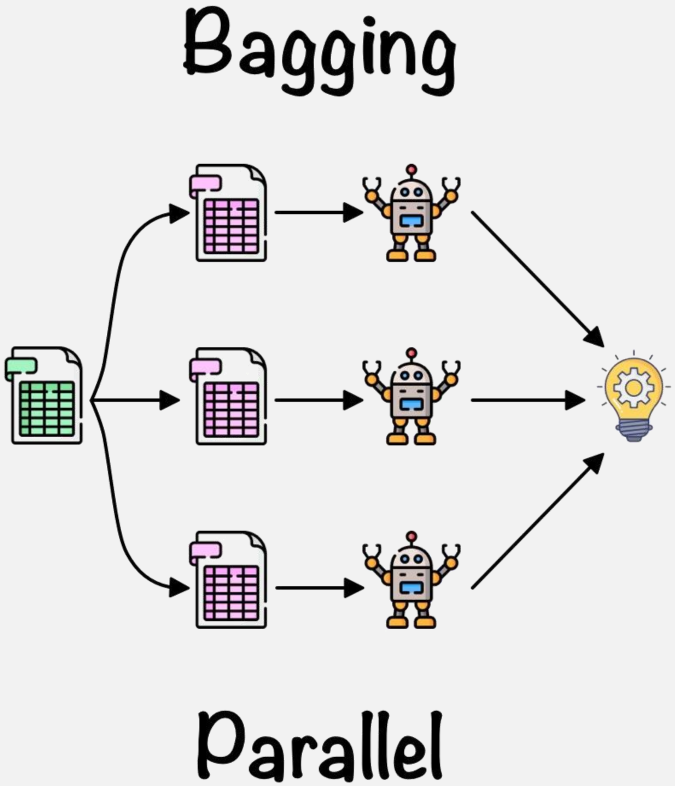
- **Training**: train several models as normal
  - Decision trees, NB, SVMs, whatever you want!
  - Each classifier gets full training data
- **Prediction**: Each model votes, majority (or average) wins



Example: Default Data

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import VotingClassifier

clf1 = DecisionTreeClassifier(max_depth=4)
clf2 = KNeighborsClassifier(n_neighbors=7)
clf3 = SVC(kernel='rbf', probability=True, gamma='scale')

classifiers = [('DT', clf1), ('KNN', clf2), ('SVM', clf3)]

cclf = VotingClassifier(estimators=classifiers, voting='soft', weights=[2, 1, 2])

cclf = cclf.fit(X_train, y_train)
```

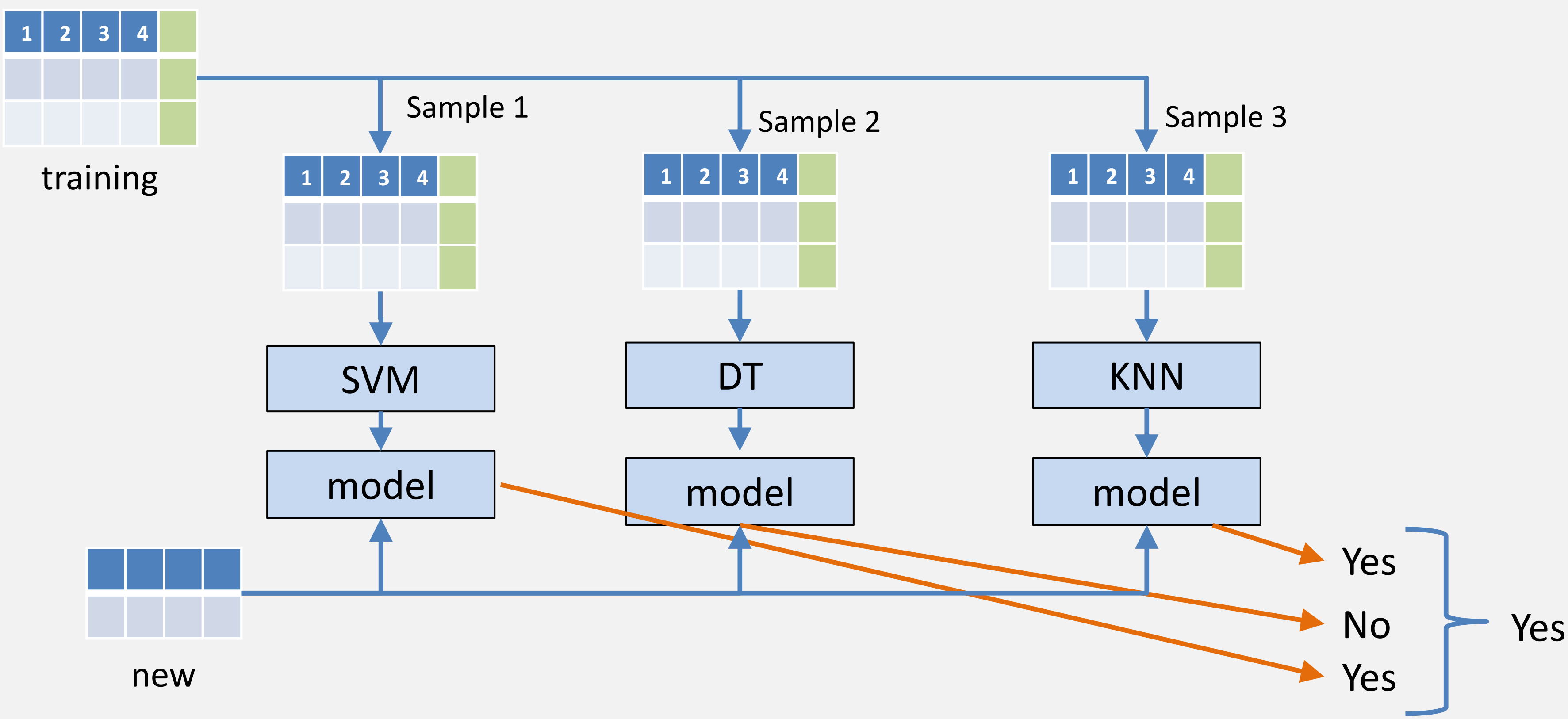
BAGGING



Source: https://towardsdatascience.com/ensemble-learning-bagging-boosting-3098079e5422

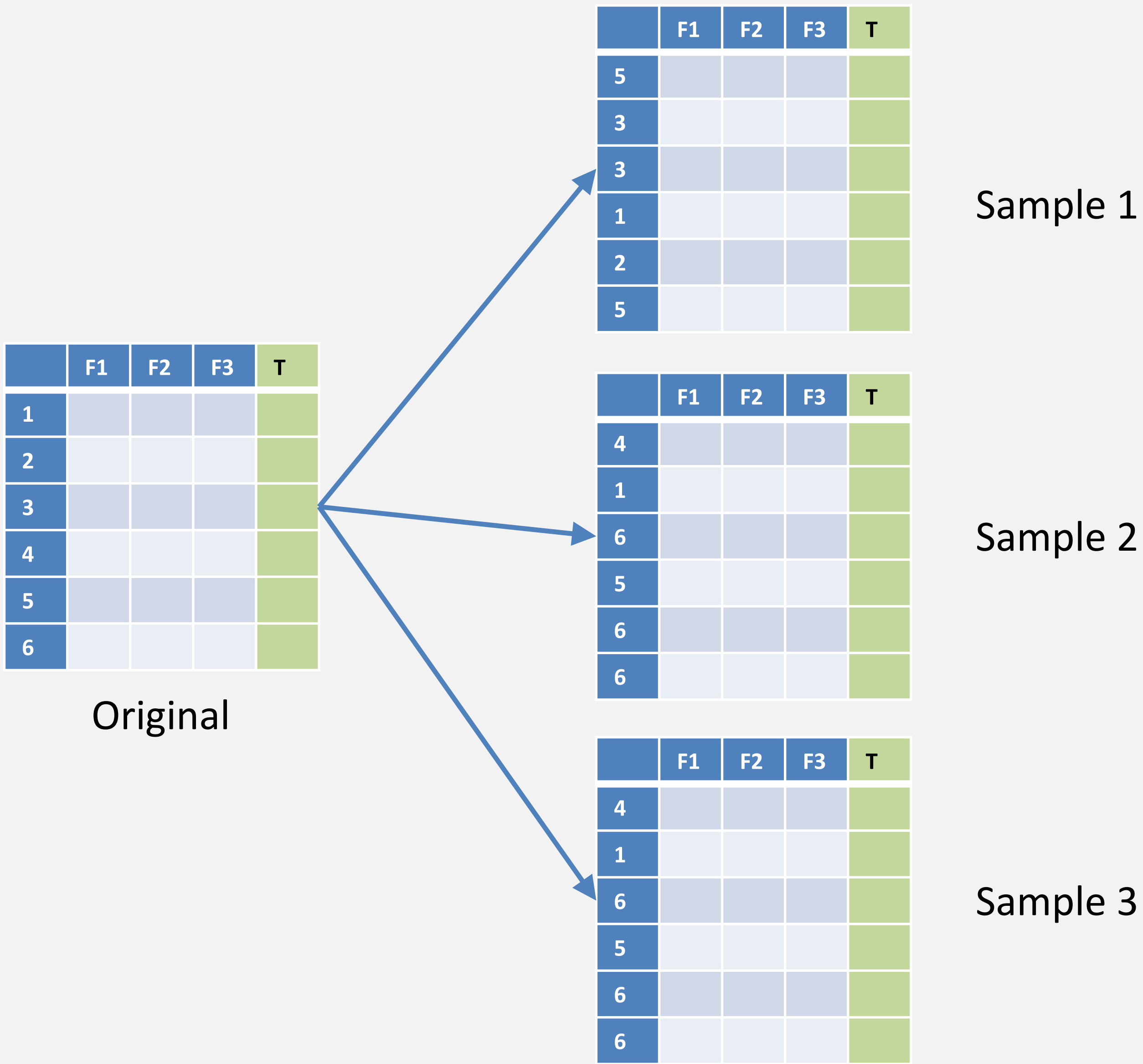
Bagging

- ***Bagging*** is the same as committees, except:
  - Instead of getting the full training data, each model only gets a ***bootstrap sample*** of the training data
  - Bootstrap described on next slide
- Popular examples: Random Forests, Extra Trees



What is a Bootstrap Sample?

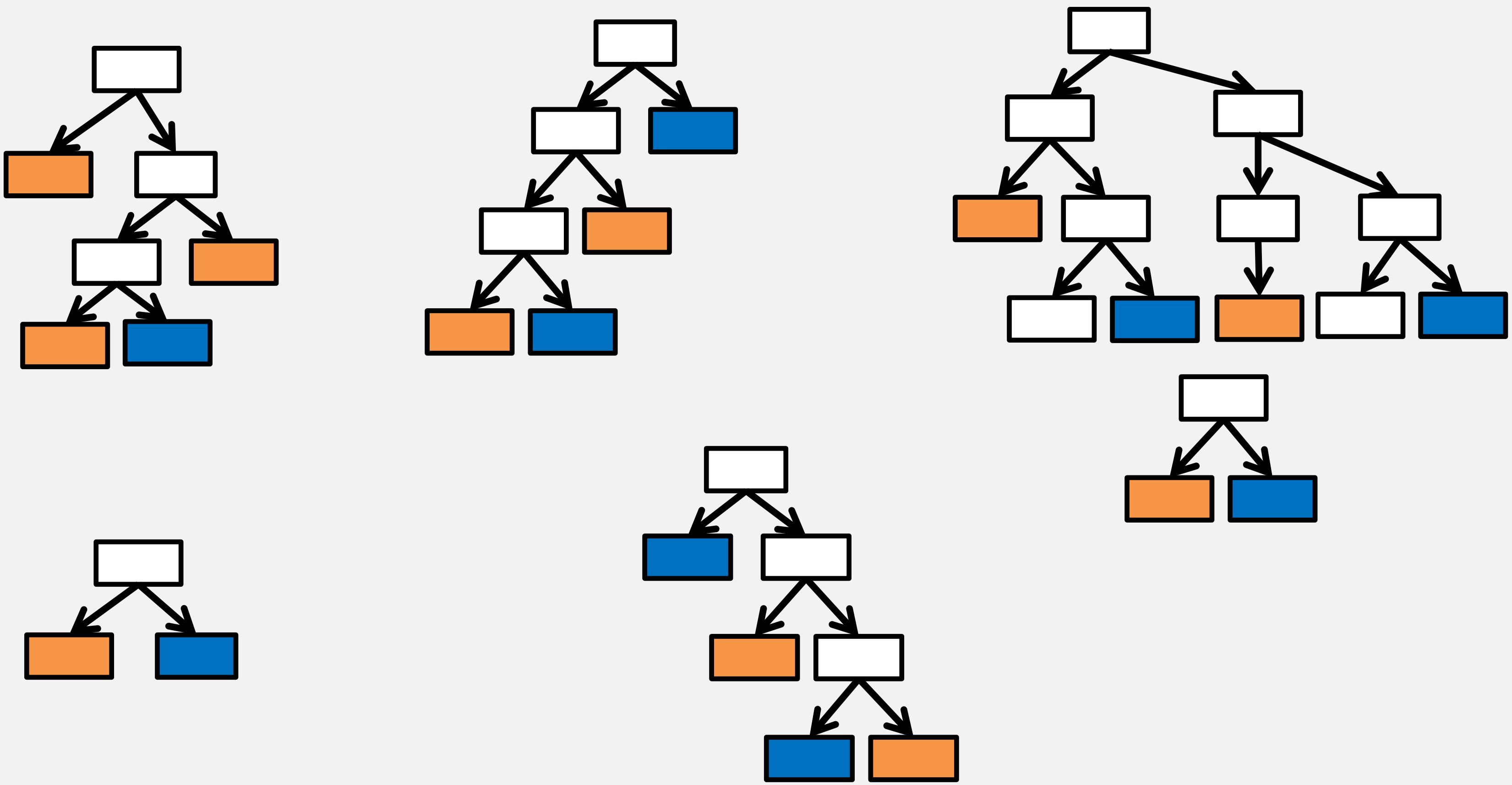
Bootstrap sampling: random with replacement



From mlxtend

Bagging Properties

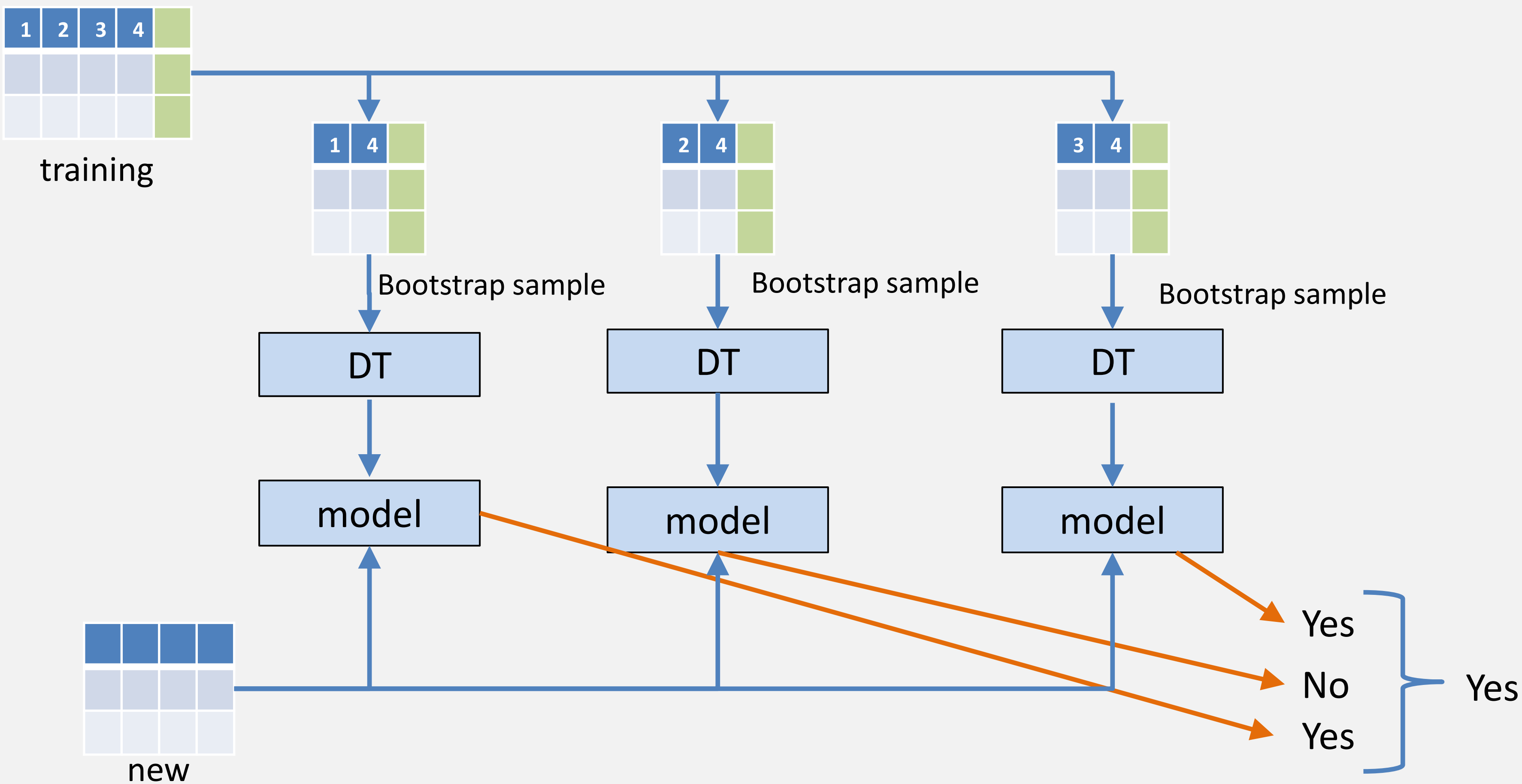
- Helps to decrease variance
  - E.g., DTs
- Fast because model training can be parallel
- Almost always helps



Random Forests

Random Forests are a bag of trees, but also:

- Each DT gets a random subset of features
- Why? Want to create trees that are not correlated with each other



Example: Default Data

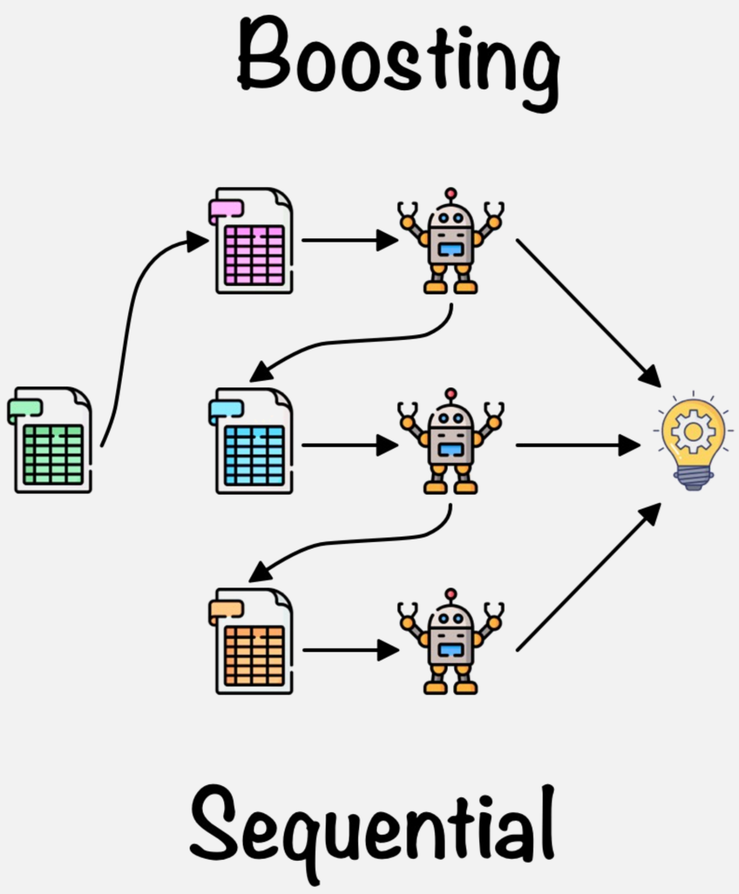
```
from sklearn.ensemble import RandomForestClassifier

clf_rf = RandomForestClassifier(
    n_estimators=100, max_depth=None, min_samples_split=2, random_state=0)
clf_rf.fit(X_train, y_train)
```

Main RF Hyperparameters

Name	Description	Default	Uncle Steve's Recommendation
n_estimators	Number of trees	100	Set to big number; don't tune
max_depth	Max depth of each tree	None	Don't tune; tune other hyperparams to control size
max_features	How many features each tree sees	'sqrt'	Don't tune; default is good
max_samples	% of instances each tree sees	None	Usually want 0.5 – 0.7. OK to tune
min_samples_split	Min number of instances in node to consider splitting	2	Higher = less overfitting. Good to tune

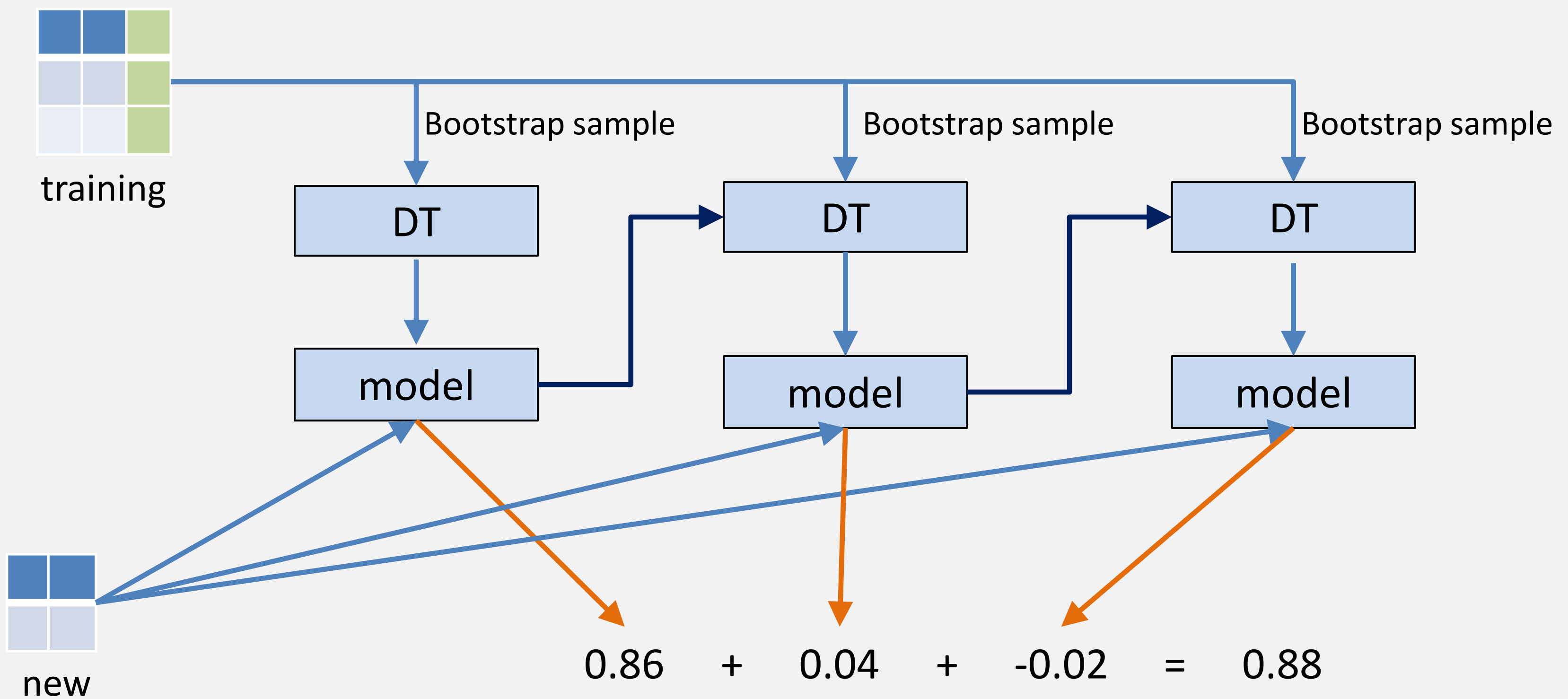
- Other hyperparameters exist, but these are the main ones

BOOSTING

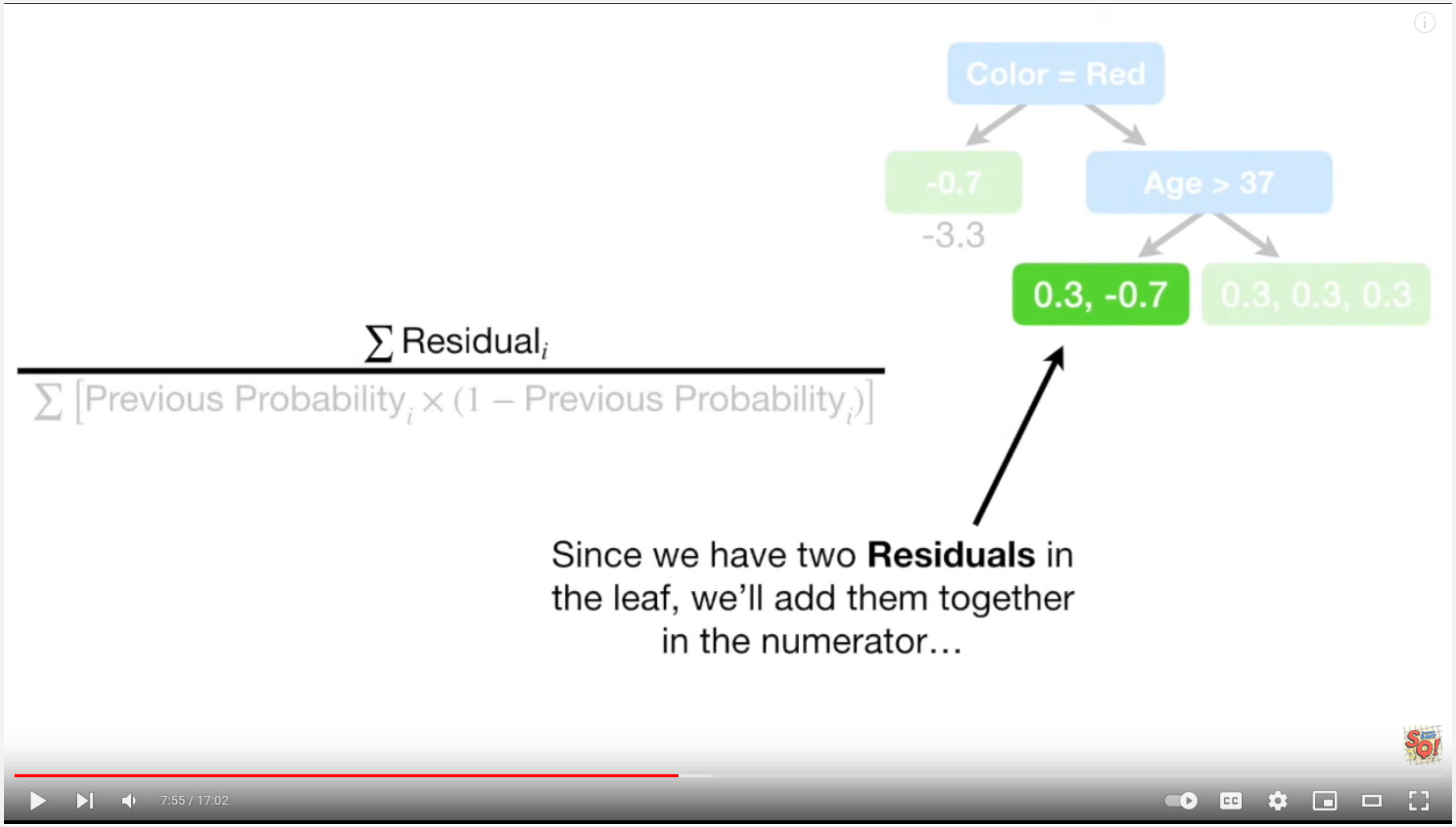


Boosting

- ***Boosting***: train models sequentially to improve previous models
- Very popular! Great results in many domains
- Variants:
  - Adaptive Boosting (e.g., AdaBoost): Original, but not the best anymore
  - Gradient Boosting (e.g., XGBoost): Very popular! Great results



StatQuest



- [4-part series on Gradient Boosting]()

XGBoost

- Popular gradient boosting algorithm from DMLC at CMU
- But more than just a boosting algorithm!
- Key features:
  - Built-in regularization
  - Memory efficient
  - Parallel/Distributed Learning
  - Sparsity-Aware Split Finding
  - Supports missing values
  - Weighted Quantile Sketch
  - …

LightGBM (LGBM)

- Gradient boosting package from Microsoft
- Key Features:
  - Faster
  - Lower memory usage
  - Better accuracy (?)
  - Support for parallel and GPU learning
  - Built-in support for categorical features

Catboost

- Gradient boosting package from Yandex
- Key Features:
  - Support for categorical features
  - Support for textual features
  - Less hyperparameter tuning required

Main LGBM Hyperparameters

Name	Description	Default	Uncle Steve's Recommendation
n_estimators	Number of trees	100	Set to medium number; don't tune
learning_rate	Amount each tree contributes	0.1	smaller -> bigger n_estimators larger -> smaller n_estimators Good to tune
num_leaves	Max number of leaves in one tree	31	Lower = less overfitting. Good to tune
max_depth	Max depth of each tree	-1	Lower = less overfitting. Good to tune
min_samples_leaf	Min num of instances in leaf	20	Higher = less overfitting. Good to tune
max_bin	Max number of bins to bucket features in	255	Lower = less overfitting. Good to tune
lambda_l1, lambda_l2	L1 and L2 regularization	0.0	Higher = less overfitting. Good to tune

- Many other hyperparameters exist that can be tuned
- https://lightgbm.readthedocs.io/en/latest/Parameters.html

Uncle Steve's Ultimate Boosting Comparison

dmlc XGBoost LightGBM Yandex CatBoost

	Adaboost	XGBoost	LightGBM	Catboost
Developer	Freund and Schapire	Tianqi Chen (CMU)	Microsoft	Yandex
Initial Release	1997	2014	2016	2017
Base learner	Stumps	Trees	Trees	Trees
Uses gradients?	No	Yes	Yes	Yes
Parallel learning?	No	Yes	Yes	Yes
GPU learning?	No	Yes	Yes	Yes
Handles categorical internally?	No	No	Yes	Yes
Grows trees via:	Levels	Levels	Leaves	Levels
Built-in regularization	No	Yes	Yes	Yes

- All help decrease bias
- Prone to overfitting
- More hyperparameters compared to Random Forests

Resources

- slides_ensemble.ipynb
- slides_ensemble_study.ipynb

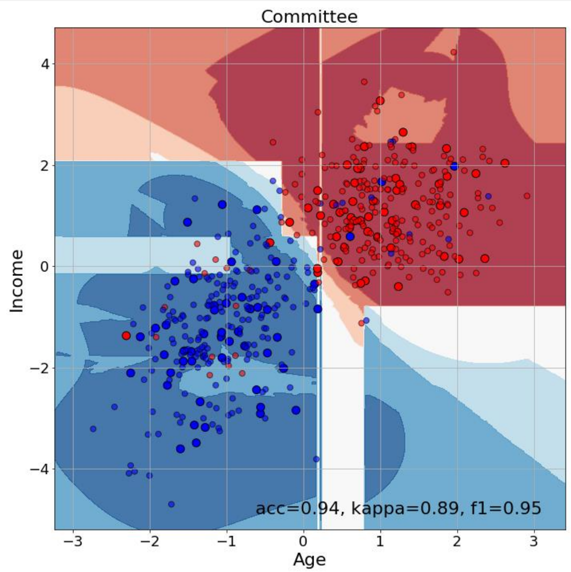
SUMMARY

Summary

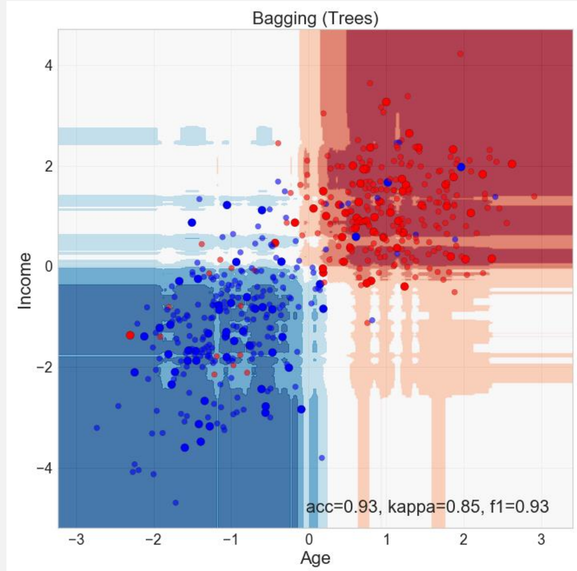
Ensembles: Combining classifiers to increase performance

- **Committee**: parallel model building with full training data
  - Not used much in practice
- **Bagging**: parallel model building with bootstrap samples
  - Great choice overall
- **Boosting**: sequential models predict errors of previous models
  - Best choice, but requires more tuning and watch out for overfitting

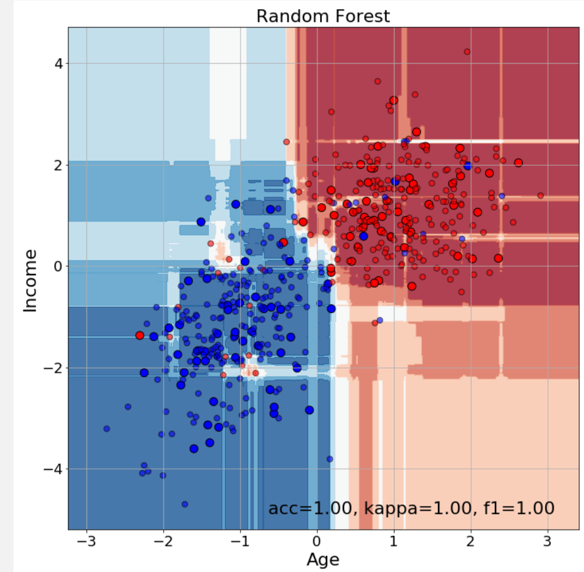
Some Art



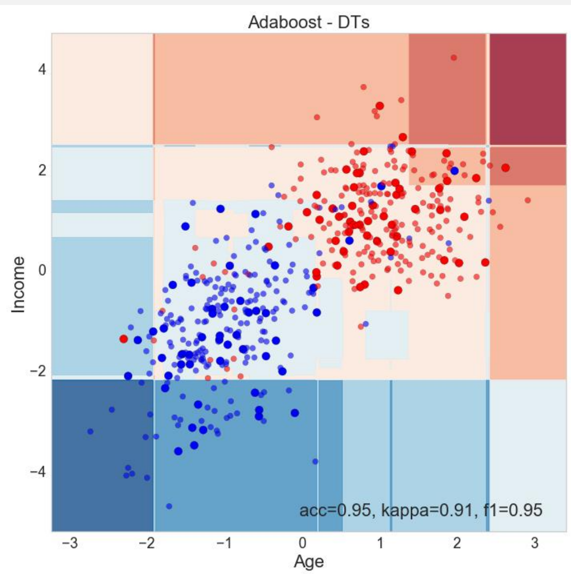
Committee
Income
Age
acc=0.94, kappa=0.89, f1=0.95



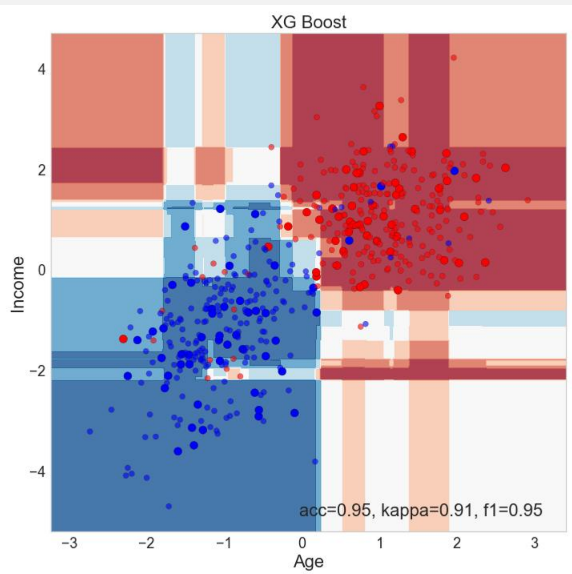
Bagging (Trees)
Income
Age
acc=0.93, kappa=0.85, f1=0.93



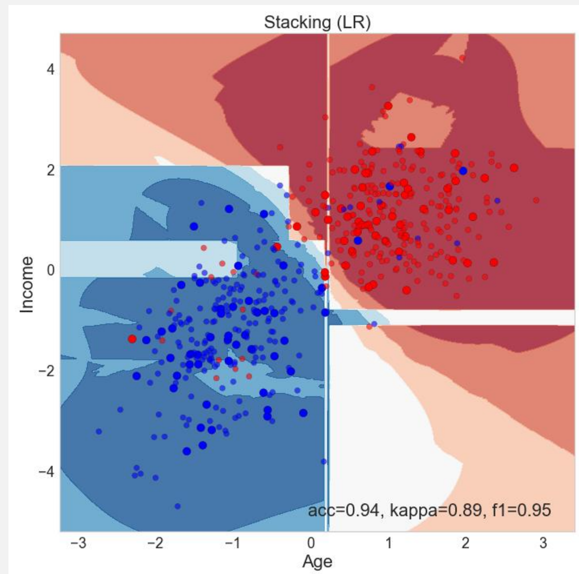
Random Forest
Income
Age
acc=1.00, kappa=1.00, f1=1.00



Adaboost - DTs
Income
Age
acc=0.95, kappa=0.91, f1=0.95



XG Boost
Income
Age
acc=0.95, kappa=0.91, f1=0.95



Stacking (LR)
Income
Age
acc=0.94, kappa=0.89, f1=0.95

APPENDIX

Feature Importance

- RFs have better performance than a single decision tree
- But, RFs are harder to interpret
  - Hundreds of trees
  - Which features are most important to the model?
- Can still get an overall summary of the importance of each feature using ***Feature Importance***
- Works by calculating the mean decrease in impurity for reach node that uses that feature

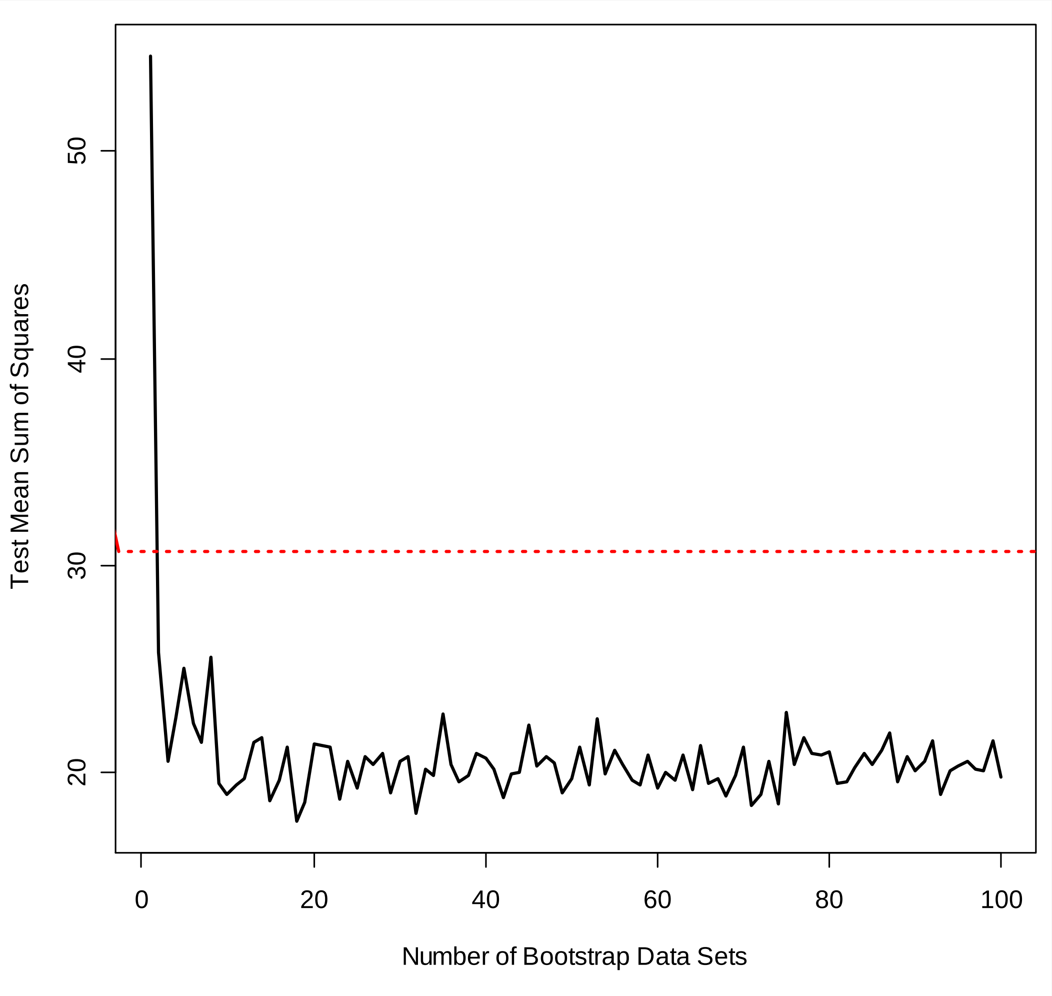
Example: Default Data

```
clf_rf.feature_importances_
```

```
array([0.5594513, 0.4405487])
```

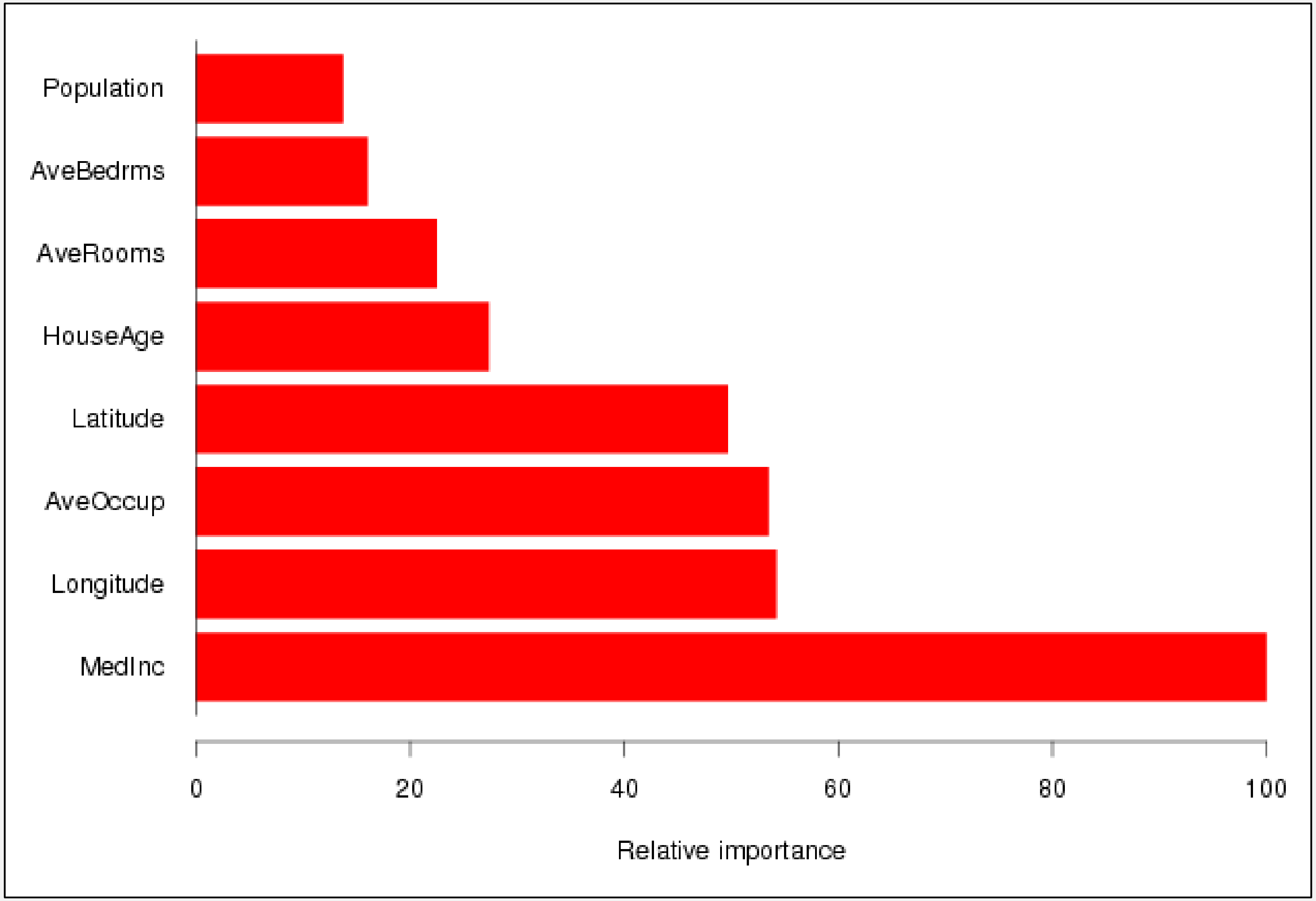
Example: Housing Data

- Red line is performance of a single tree
- Black line is performance of tree bagging



Example: Housing Data

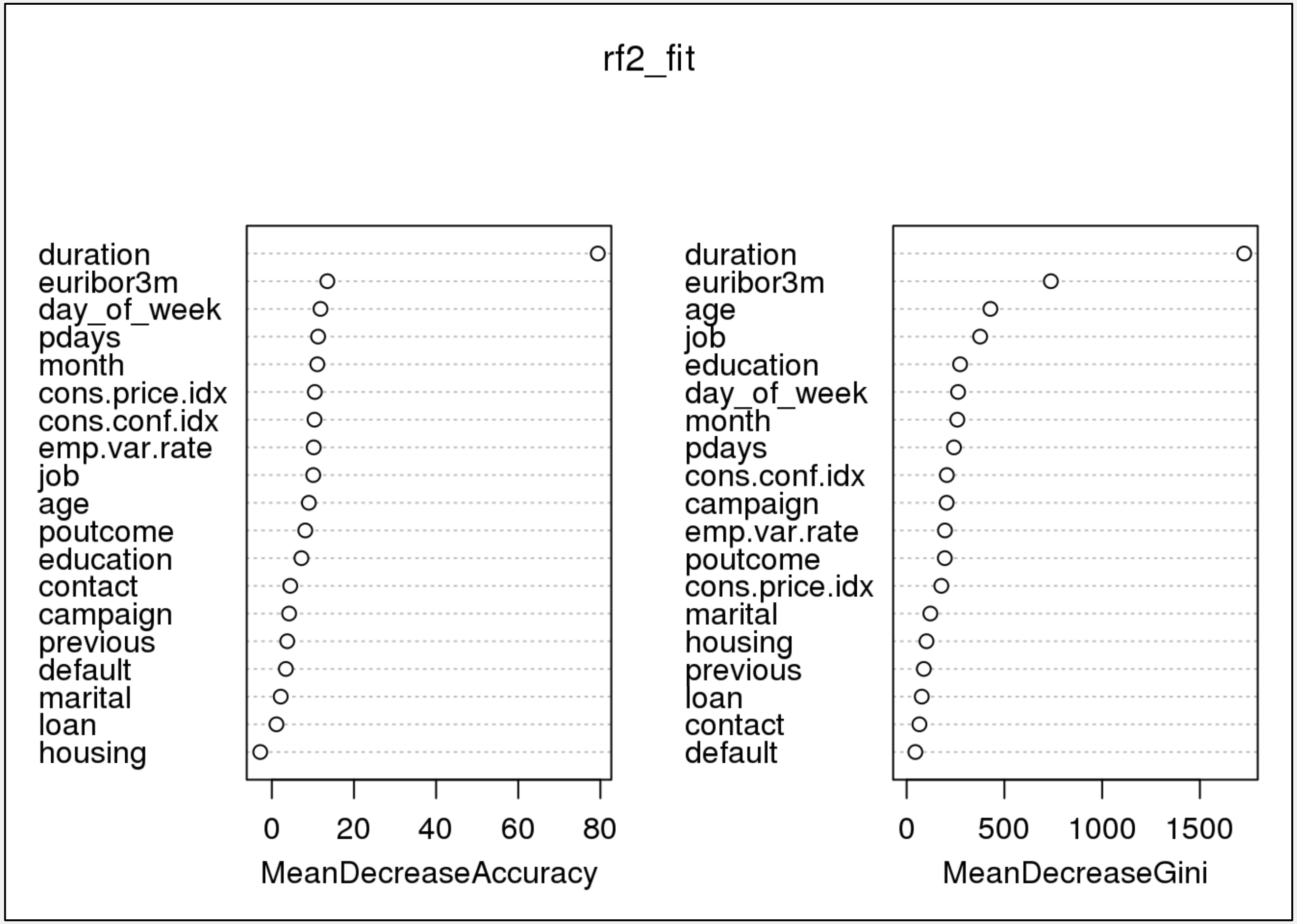
- Median Income is by far the most important variable.
- Longitude, Latitude and Average occupancy are the next most important.



Example

```
rf2_fit = randomForest(formula, data=train, mtry=3, ntree=100, importance=TRUE)
```

```
varImpPlot(rf2_fit)
```



Example

```
rf2_pred = predict(rf2_fit, test, type="class")
caret::confusionMatrix(data=rf2_pred,
            reference=actual, positive=positive, dnn=c("Predicted", "Actual"))
```

```
## Confusion Matrix and Statistics
##
##          Actual
## Predicted   no  yes
##       no  7102  464
##       yes  207  464
##
##                Accuracy : 0.9185
##                  95% CI : (0.9124, 0.9244)
##     No Information Rate : 0.8873
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5365
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.50000
##             Specificity : 0.97168
##          Pos Pred Value : 0.69151
##          Neg Pred Value : 0.93867
##              Prevalence : 0.11266
##          Detection Rate : 0.05633
##    Detection Prevalence : 0.08146
##       Balanced Accuracy : 0.73584
##
##        'Positive' Class : yes
##
```

Example

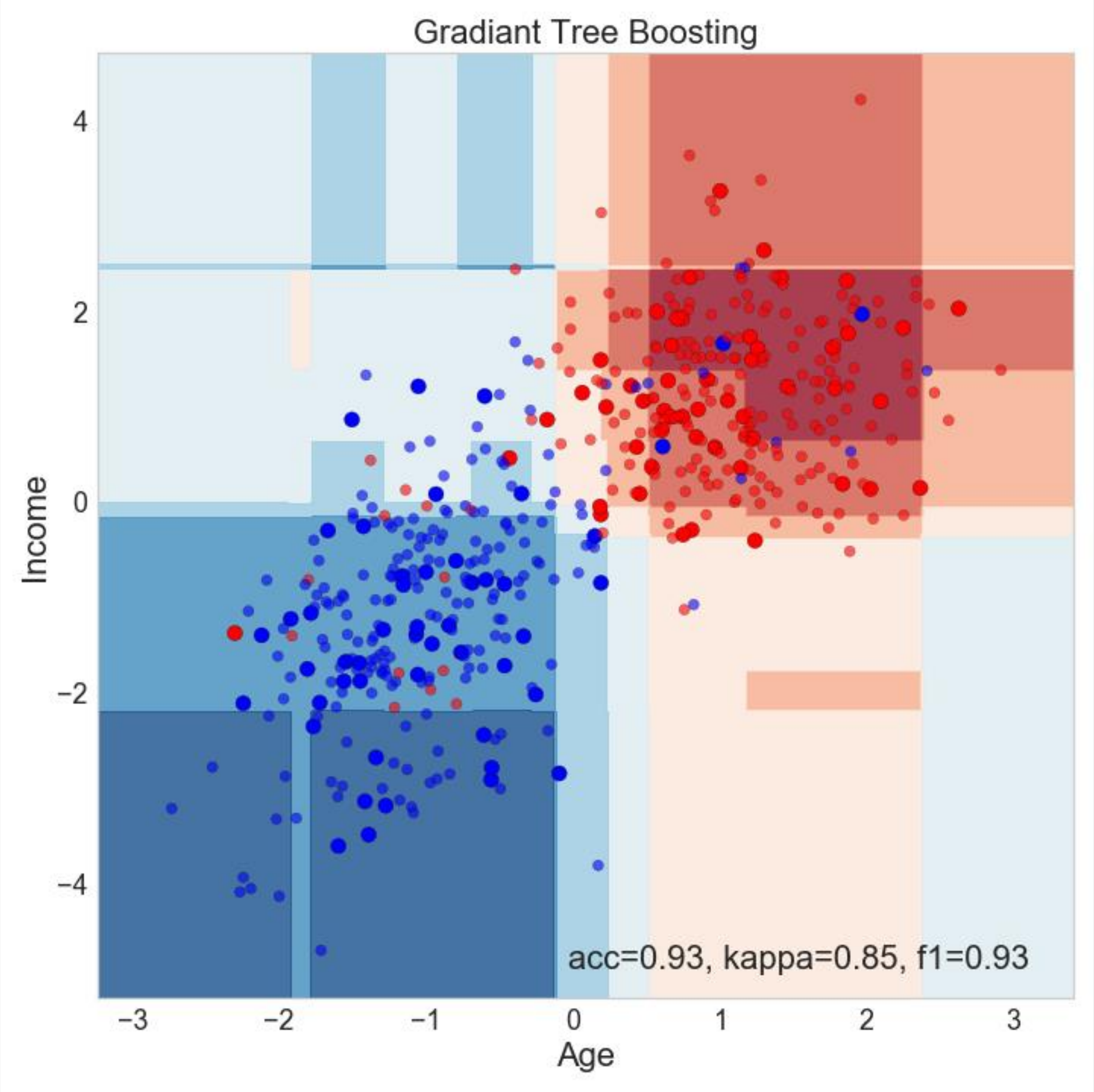
```
library(fastAdaboost)
boost = adaboost(formula, data=train, nIter=20)
boost_pred = predict(boost, newdata=test)
caret::confusionMatrix(data=boost_pred$class,
            reference=actual, positive=positive, dnn=c("Predicted", "Actual"))
```

```
## Confusion Matrix and Statistics
##
##          Actual
## Predicted   no  yes
##       no  6992  431
##       yes  317  497
##
##                Accuracy : 0.9092
##                  95% CI : (0.9028, 0.9153)
##     No Information Rate : 0.8873
##     P-Value [Acc > NIR] : 5.845e-11
##
##                   Kappa : 0.5201
##  Mcnemar's Test P-Value : 3.601e-05
##
##             Sensitivity : 0.53556
##             Specificity : 0.95663
##          Pos Pred Value : 0.61057
##          Neg Pred Value : 0.94194
##              Prevalence : 0.11266
##          Detection Rate : 0.06034
##    Detection Prevalence : 0.09882
##       Balanced Accuracy : 0.74609
##
##        'Positive' Class : yes
##
```

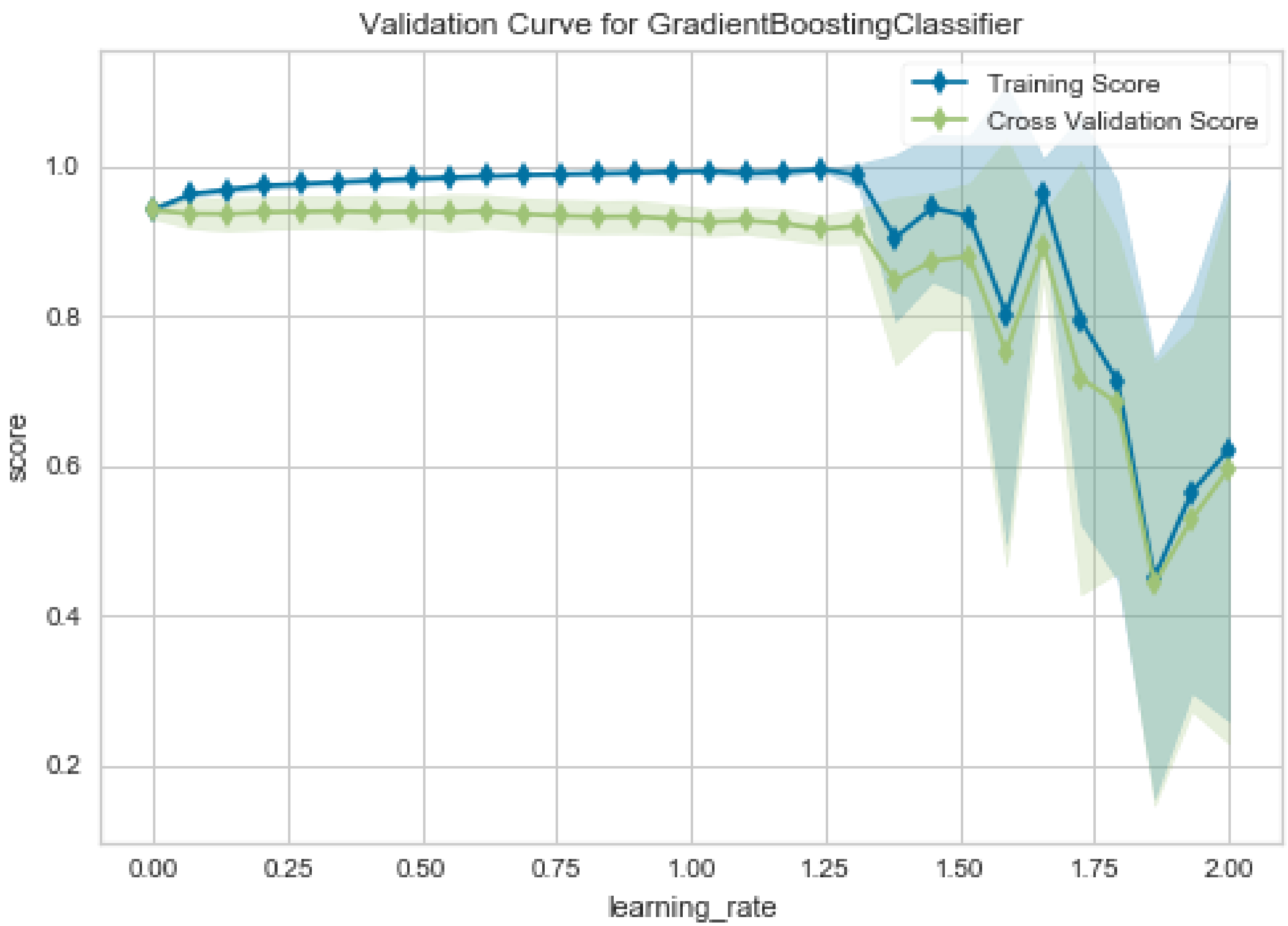
Example: Default Data

```
from sklearn.ensemble import GradientBoostingClassifier

clf_gt = GradientBoostingClassifier(
    n_estimators=100, learning_rate=0.1, max_depth=1, max_features=1,
    random_state=0)
clf_gt.fit(X_train, y_train)
```

Example: Default Data



Example: Default Data



Example

```
library(xgboost)
xgboost_fit <- xgboost(data = train_data, label=train_label,
            nround=20, verbose=2, objective="binary:logistic")
xgboost_pred = predict(xgboost_fit, newdata=test_data)
xgboost_pred2 = as.factor(ifelse(xgboost_pred > 0.5, "yes", "no"))
caret::confusionMatrix(data=xgboost_pred2,
            reference=actual, positive=positive, dnn=c("Predicted", "Actual"))
```

```
## Confusion Matrix and Statistics
##
##          Actual
## Predicted   no  yes
##       no  6992  431
##       yes  317  497
##
##                Accuracy : 0.9092
##                  95% CI : (0.9028, 0.9153)
##     No Information Rate : 0.8873
##     P-Value [Acc > NIR] : 5.845e-11
##
##                   Kappa : 0.5201
##  Mcnemar's Test P-Value : 3.601e-05
##
##             Sensitivity : 0.53556
##             Specificity : 0.95663
##          Pos Pred Value : 0.61057
##          Neg Pred Value : 0.94194
##              Prevalence : 0.11266
##          Detection Rate : 0.06034
##    Detection Prevalence : 0.09882
##       Balanced Accuracy : 0.74609
##
##        'Positive' Class : yes
##
```

Example

```
ctrl = trainControl(
            method="boot", number=10,
            savePredictions="final",
            classProbs=TRUE,
            index=createResample(train$bought, 10),
            summaryFunction=twoClassSummary,
            allowParallel = TRUE)

model_list <- caretList(
            bought~., data=train, trControl=ctrl,
            methodList=c("pls", "rpart") )

stack_fit <- caretStack(
  model_list, method="glm", metric="ROC",
  trControl=trainControl(
    method="boot",
    number=10,
    savePredictions="final",
    classProbs=TRUE,
    summaryFunction=twoClassSummary))
```

Example

```
summary(stack_fit)
```

```
##
## Call:
## NULL
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.1899  -0.3152  -0.2775  -0.2530   2.6794
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -5.89231    0.05101 -115.52   <2e-16 ***
## pls          8.71604    0.15620   55.80   <2e-16 ***
## rpart        4.16808    0.05815   71.68   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 85824  on 121629  degrees of freedom
## Residual deviance: 56948  on 121627  degrees of freedom
## AIC: 56954
##
## Number of Fisher Scoring iterations: 6
```

Example

```
stack_pred = predict(stack_fit, test)
caret::confusionMatrix(data=stack_pred,
            reference=actual, positive=positive, dnn=c("Predicted", "Actual"))
```

```
## Confusion Matrix and Statistics
##
##          Actual
## Predicted   no  yes
##       no  7109  536
##       yes  200  392
##
##                Accuracy : 0.9106
##                  95% CI : (0.9043, 0.9167)
##     No Information Rate : 0.8873
##     P-Value [Acc > NIR] : 2.799e-12
##
##                   Kappa : 0.4692
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.42241
##             Specificity : 0.97264
##          Pos Pred Value : 0.66216
##          Neg Pred Value : 0.92989
##              Prevalence : 0.11266
##          Detection Rate : 0.04759
##    Detection Prevalence : 0.07187
##       Balanced Accuracy : 0.69753
##
##        'Positive' Class : yes
##
```

Example

```
rpart_fit <- train(formula, data = train, method="rpart", trControl = ctrl, metric="Kappa")
rpart_pred = predict(rpart_fit, test)

nb_fit <- train(formula, data = train, "naive_bayes", trControl = ctrl, metric="Kappa")
nb_pred = predict(nb_fit, test)

pls_fit <- train(formula, data = train, "pls", trControl = ctrl)
pls_pred = predict(pls_fit, test)
```

```
committee_pred = as.data.frame(
            cbind(as.character(rpart_pred), as.character(nb_pred), as.character(pls_pred)))


committee_pred$yes_count = apply(committee_pred[,1:3], 1, function(x) sum(x=="yes"))
committee_pred$no_count = apply(committee_pred[,1:3], 1, function(x) sum(x=="no"))
committee_pred$vote = factor(
            ifelse(committee_pred$yes_count >= committee_pred$no_count, "yes", "no"))
```

Example

```
head(committee_pred, n=30)
```

```
##       V1  V2  V3 yes_count no_count vote
## 1     no  no  no         0        3   no
## 2     no  no  no         0        3   no
## 3     no  no  no         0        3   no
## 4     no  no  no         0        3   no
## 5     no  no  no         0        3   no
## 6     no  no  no         0        3   no
## 7     no  no  no         0        3   no
## 8     no  no  no         0        3   no
## 9     no  no  no         0        3   no
## 10    no  no  no         0        3   no
## 11   yes  no yes         2        1  yes
## 12    no  no  no         0        3   no
## 13    no  no  no         0        3   no
## 14    no  no  no         0        3   no
## 15   yes  no yes         2        1  yes
## 16    no  no  no         0        3   no
## 17    no  no  no         0        3   no
## 18    no  no  no         0        3   no
## 19    no  no  no         0        3   no
## 20    no  no  no         0        3   no
## 21    no  no  no         0        3   no
## 22    no  no  no         0        3   no
## 23    no  no  no         0        3   no
## 24    no  no yes         1        2   no
## 25   yes  no yes         2        1  yes
## 26    no  no  no         0        3   no
## 27    no  no  no         0        3   no
## 28    no  no  no         0        3   no
## 29    no  no  no         0        3   no
## 30    no  no  no         0        3   no
```

Example

```
caret::confusionMatrix(data=rpart_pred,
            reference=actual, positive=positive, dnn=c("Predicted", "Actual"))
```

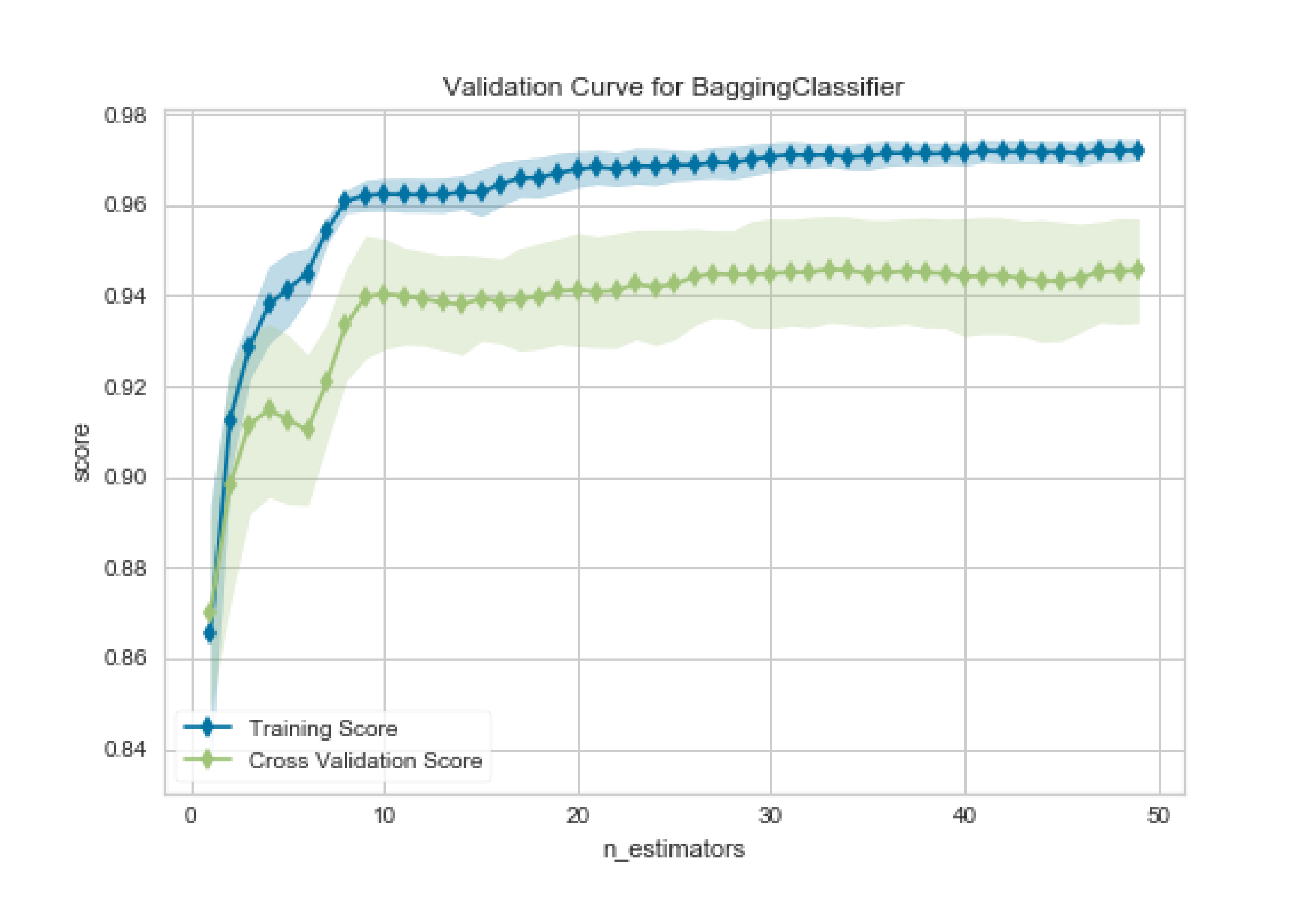
```
## Confusion Matrix and Statistics
##
##          Actual
## Predicted   no  yes
##       no  5747   76
##       yes 1562  852
##
##                Accuracy : 0.8011
##                  95% CI : (0.7924, 0.8097)
##     No Information Rate : 0.8873
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.4146
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.9181
##             Specificity : 0.7863
##          Pos Pred Value : 0.3529
##          Neg Pred Value : 0.9869
##              Prevalence : 0.1127
##          Detection Rate : 0.1034
##    Detection Prevalence : 0.2931
##       Balanced Accuracy : 0.8522
##
##        'Positive' Class : yes
##
```

Example

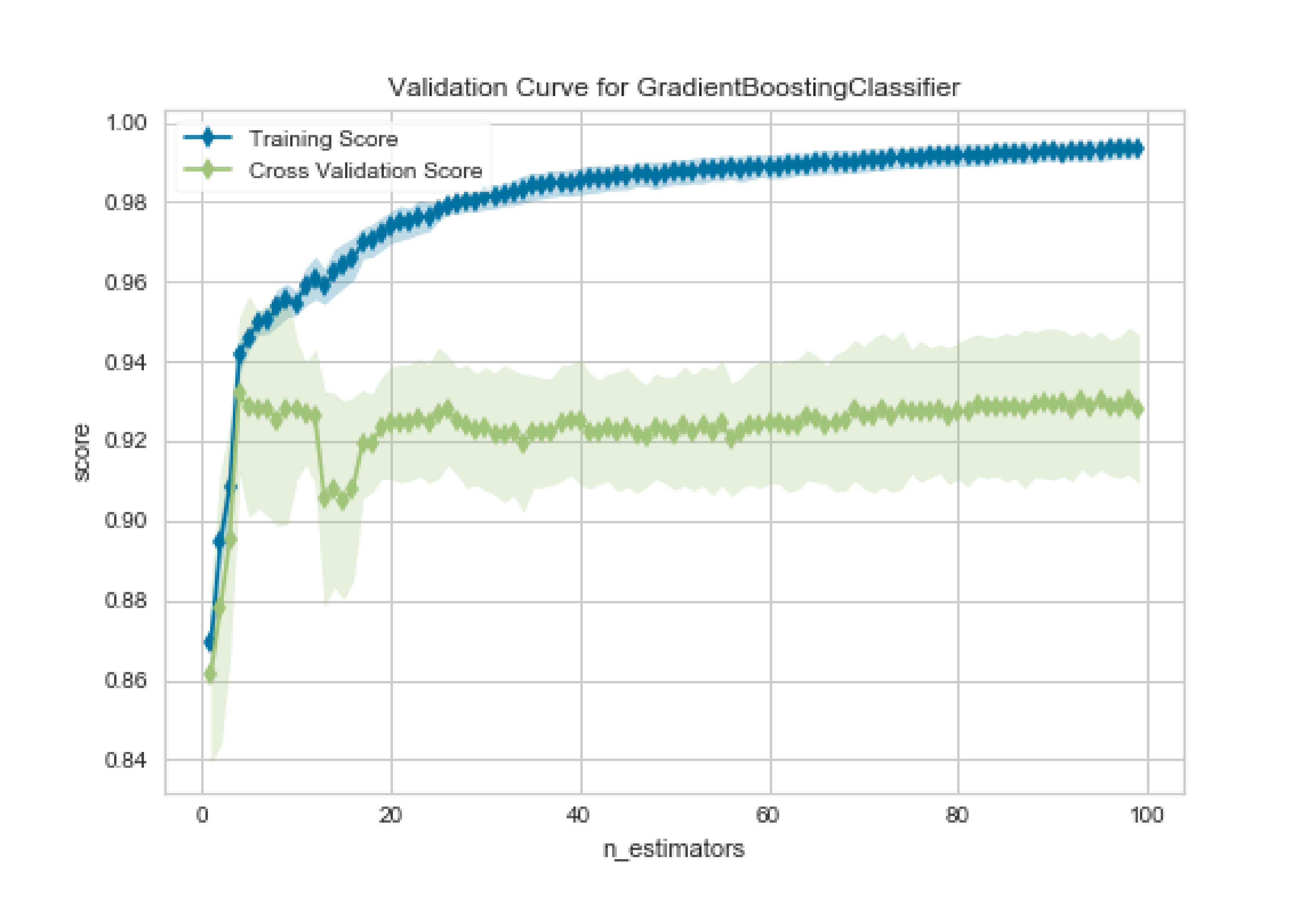
```
caret::confusionMatrix(data=committee_pred$vote,
          reference=actual, positive=positive, dnn=c("Predicted", "Actual"))
```

```
## Confusion Matrix and Statistics
##
##          Actual
## Predicted   no  yes
##       no  6420  217
##       yes  889  711
##
##                Accuracy : 0.8657
##                  95% CI : (0.8582, 0.873)
##     No Information Rate : 0.8873
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.4897
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.76616
##             Specificity : 0.87837
##          Pos Pred Value : 0.44438
##          Neg Pred Value : 0.96730
##              Prevalence : 0.11266
##          Detection Rate : 0.08632
##    Detection Prevalence : 0.19425
##       Balanced Accuracy : 0.82227
##
##        'Positive' Class : yes
##
```

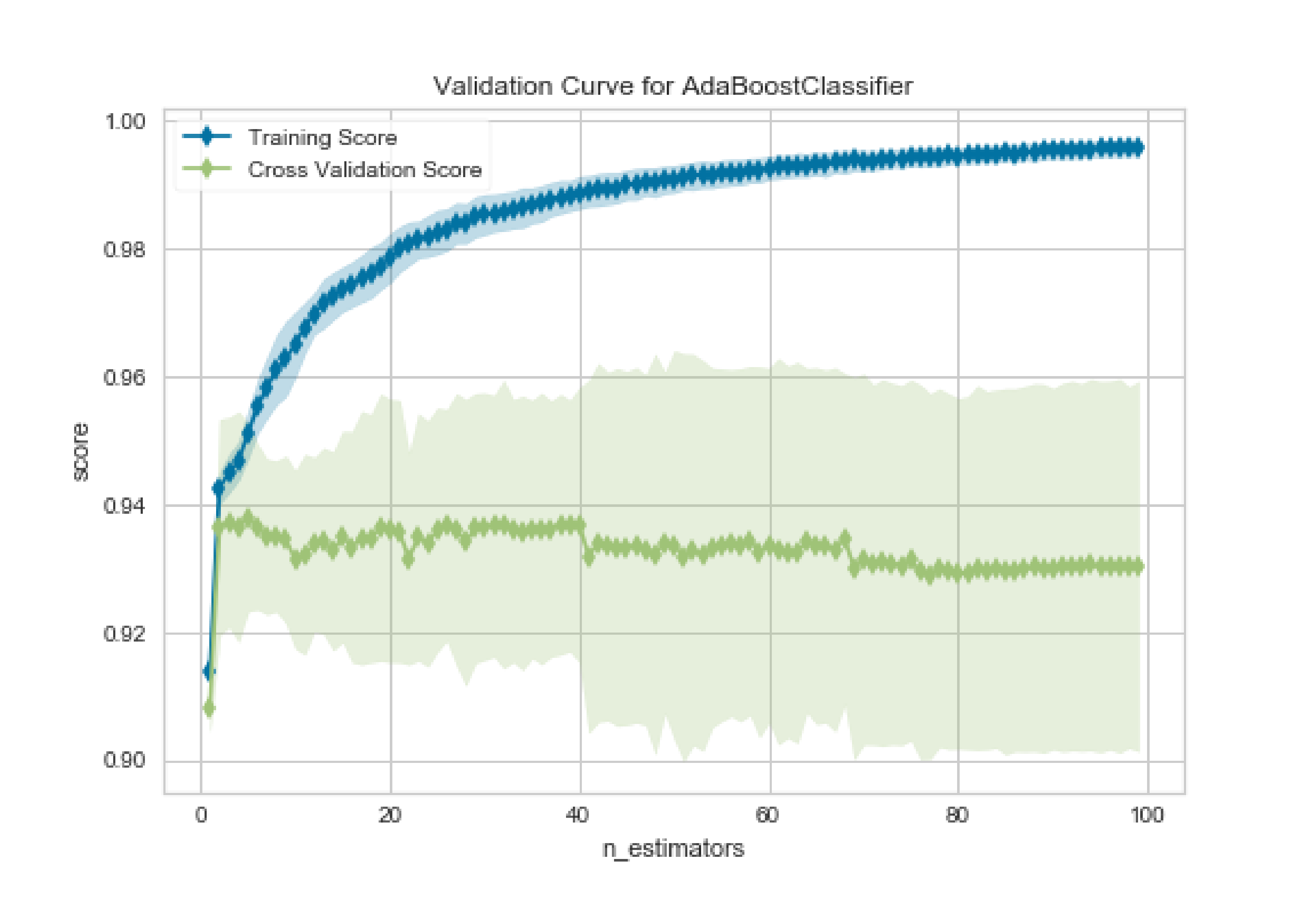
Example: Default Data



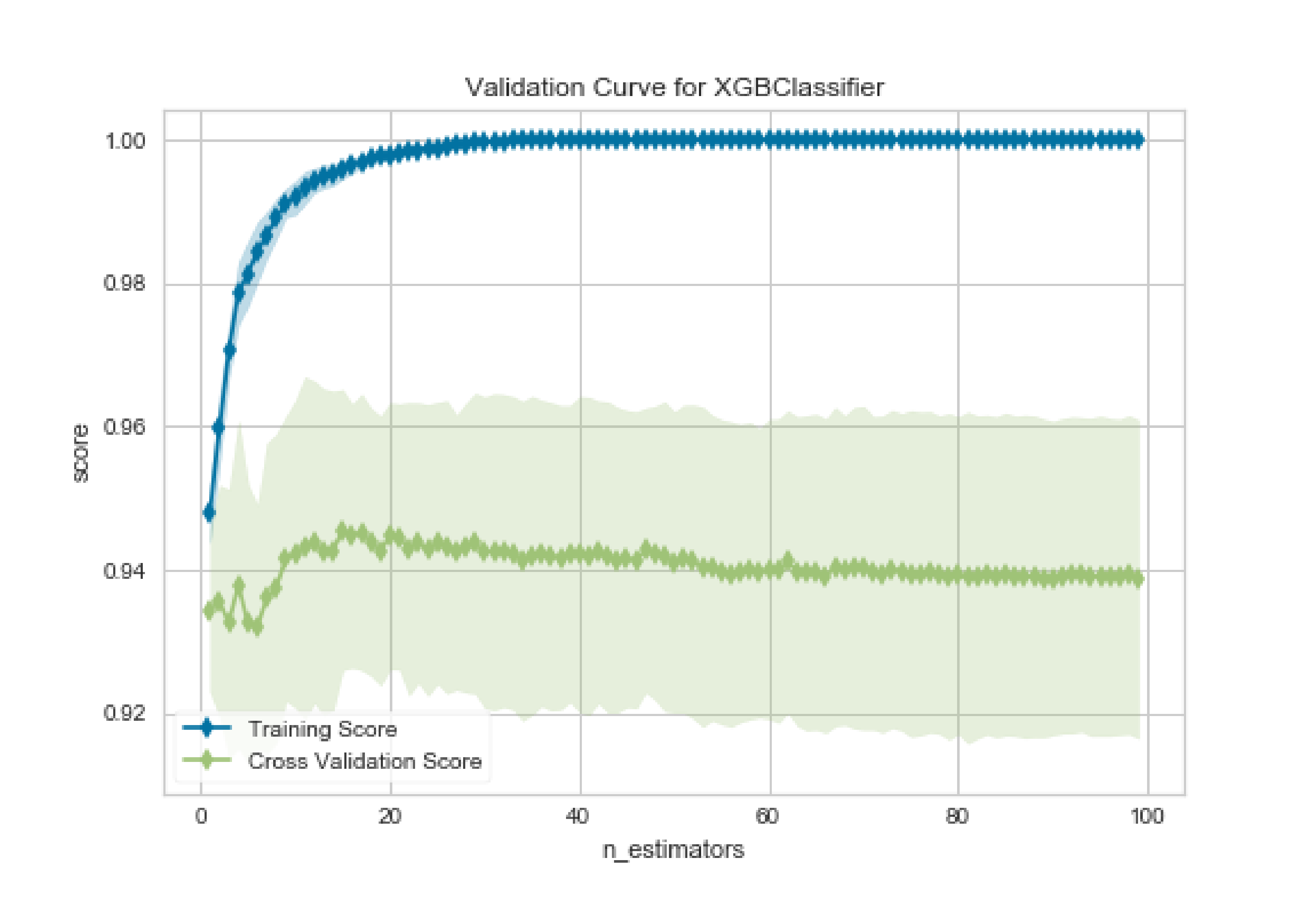
Example: Default Data



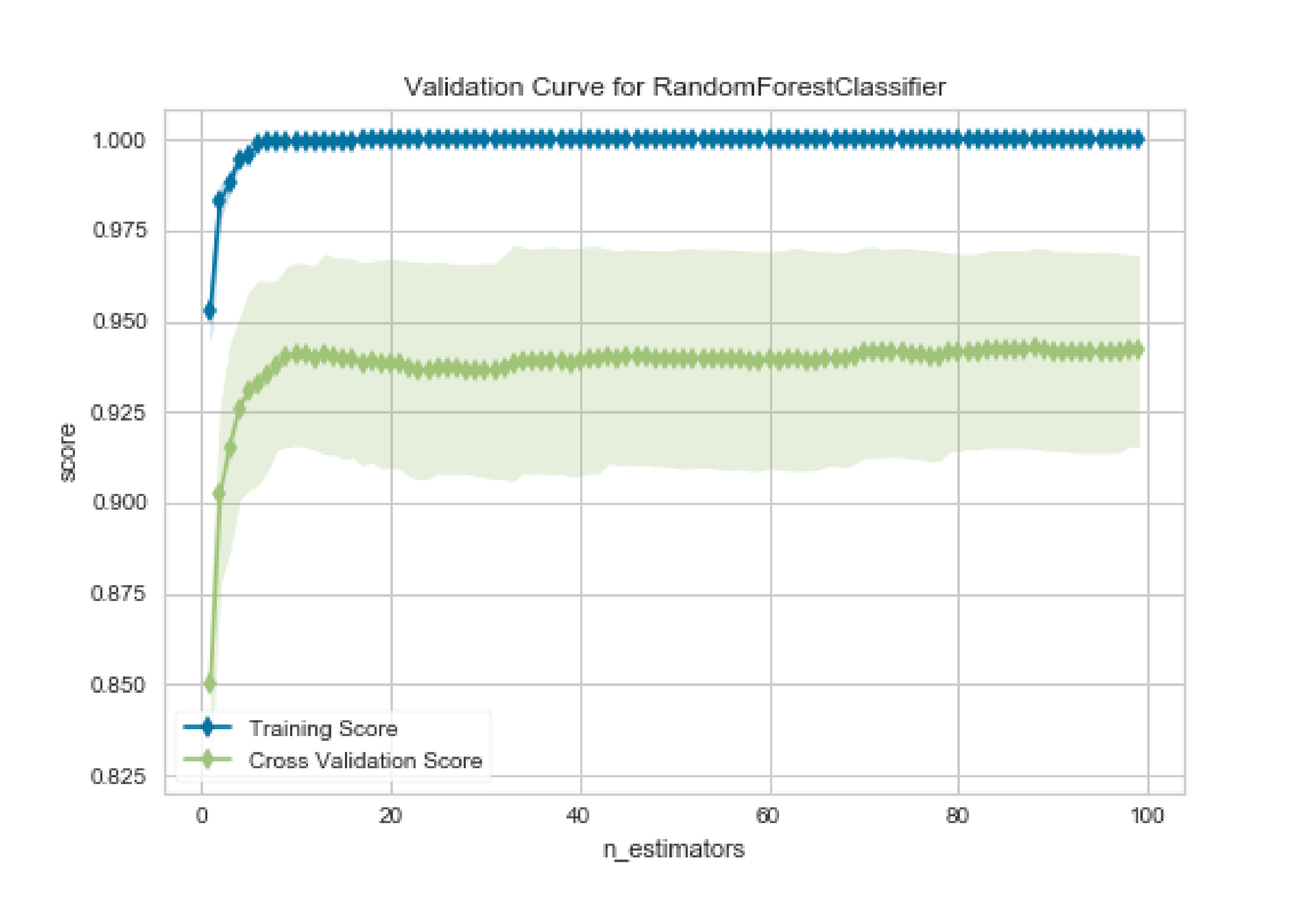
Example: Default Data



Example: Default Data



Example: Default Data



Pseudo Code

Let one class be represented with a value of +1 and the other with a value of -1

Let each sample have the same starting weight $(1/n)$

for k = *1 to* K **do**

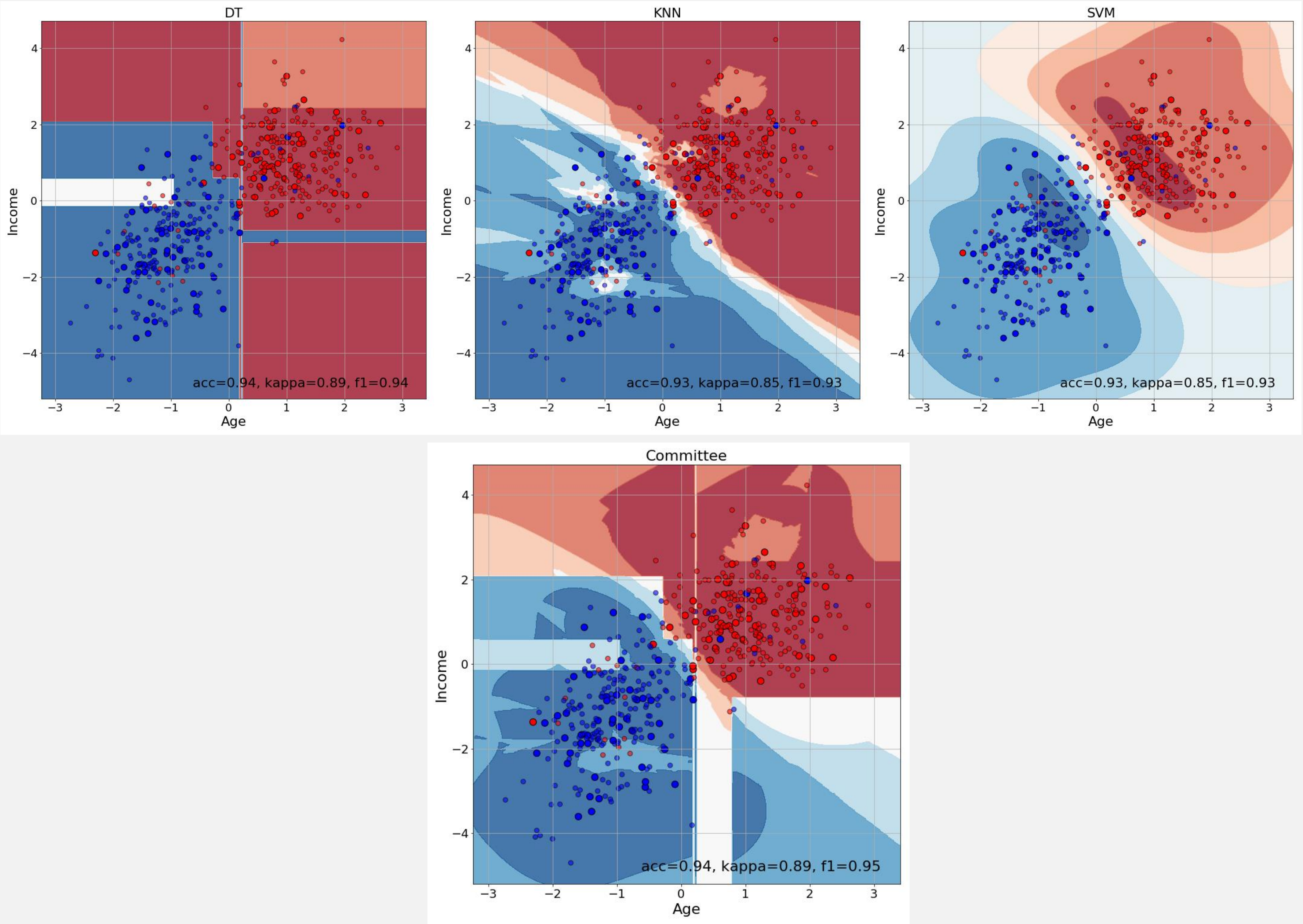
> Fit a weak classifier using the weighted samples and compute the kth model's misclassification error (err_k)
>
> Compute the kth stage value as $\ln\left((1 - err_k)/err_k\right)$.
>
> Update the sample weights giving more weight to incorrectly predicted samples and less weight to correctly predicted samples

end

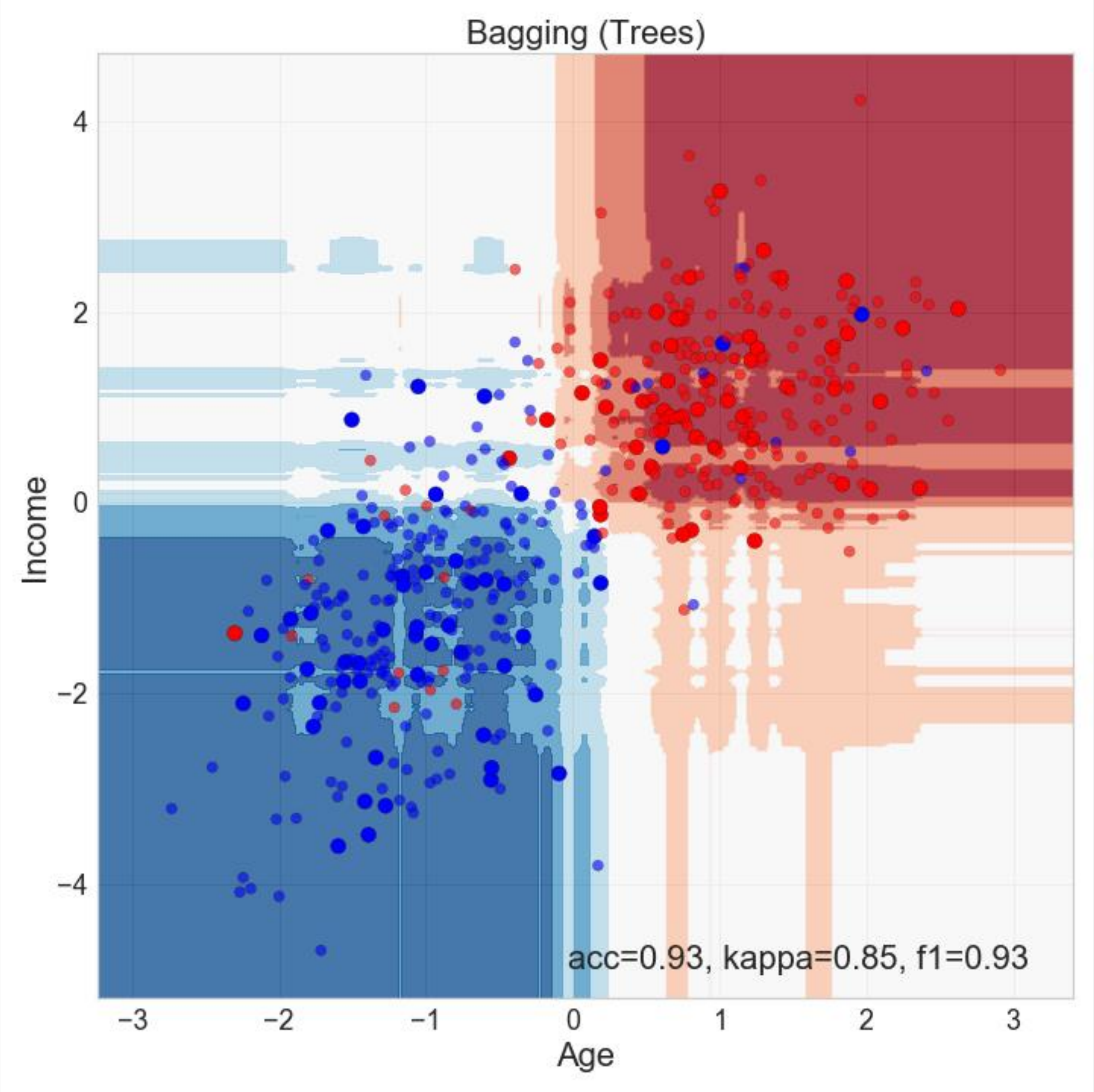
Compute the boosted classifier's prediction for each sample by multiplying the kth stage value by the kth model prediction and adding these quantities across k. If this sum is positive, then classify the sample in the +1 class, otherwise the -1 class.

Algorithm 14.2: AdaBoost algorithm for two-class problems

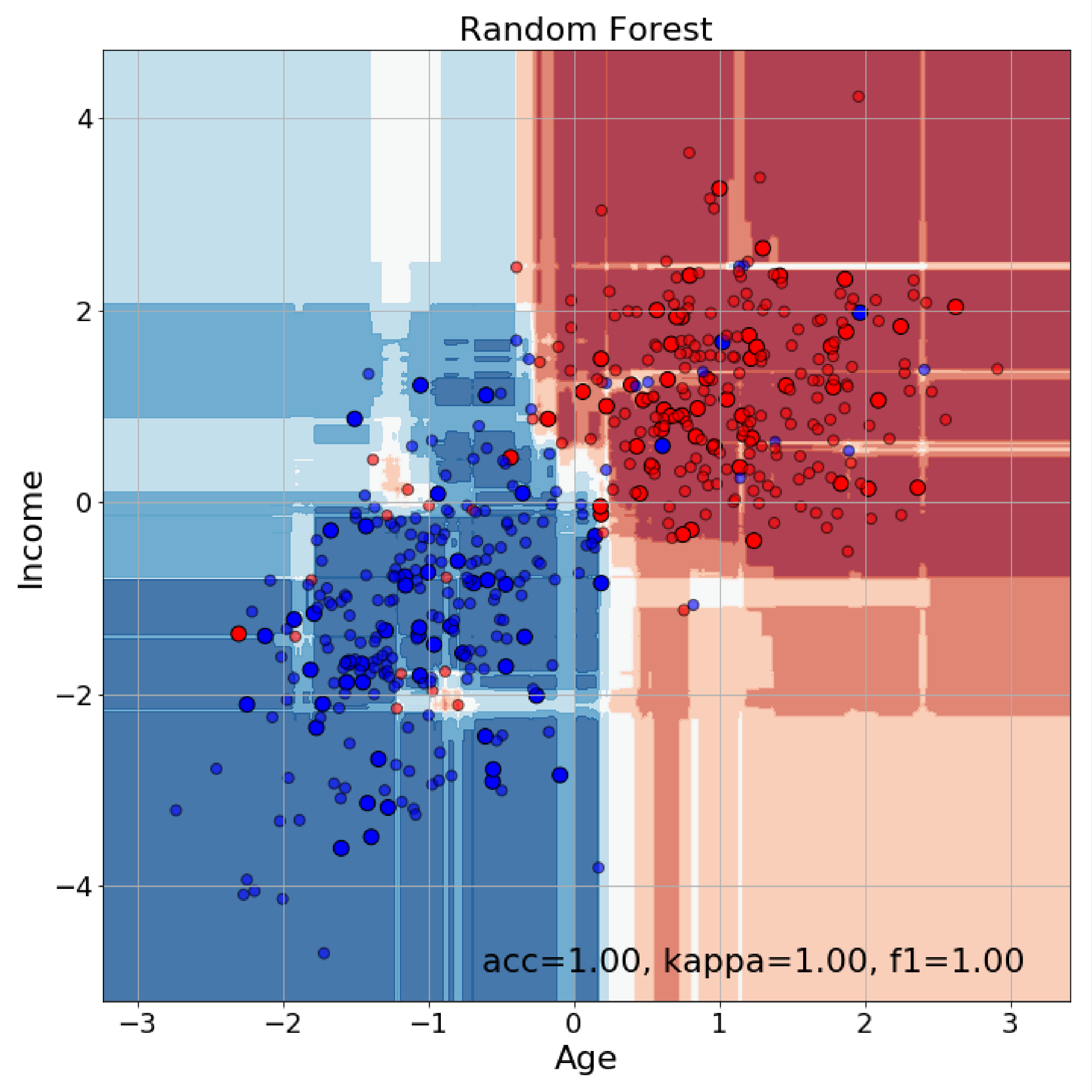
Example: Default Data



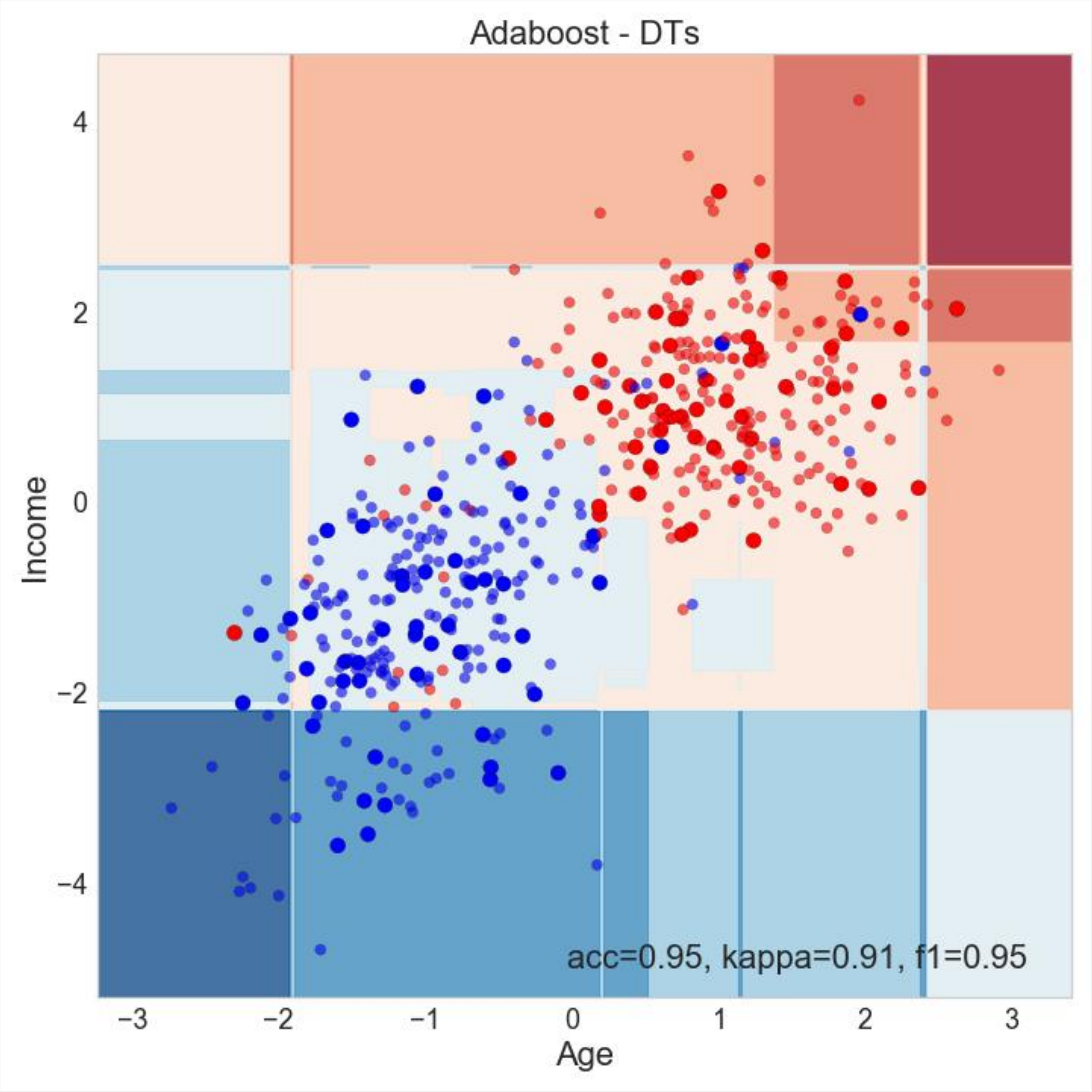
Example: Default Data



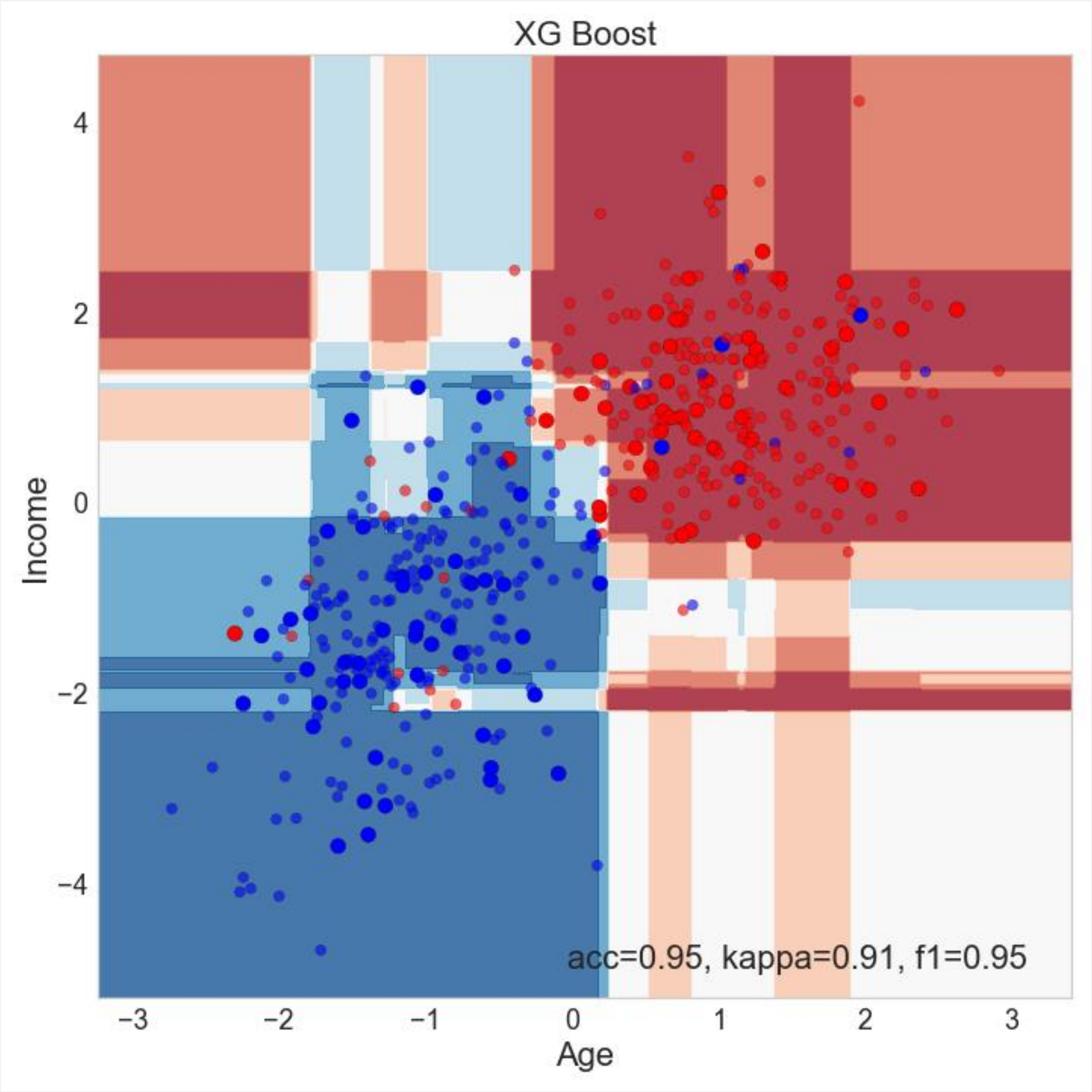
Example: Default Data



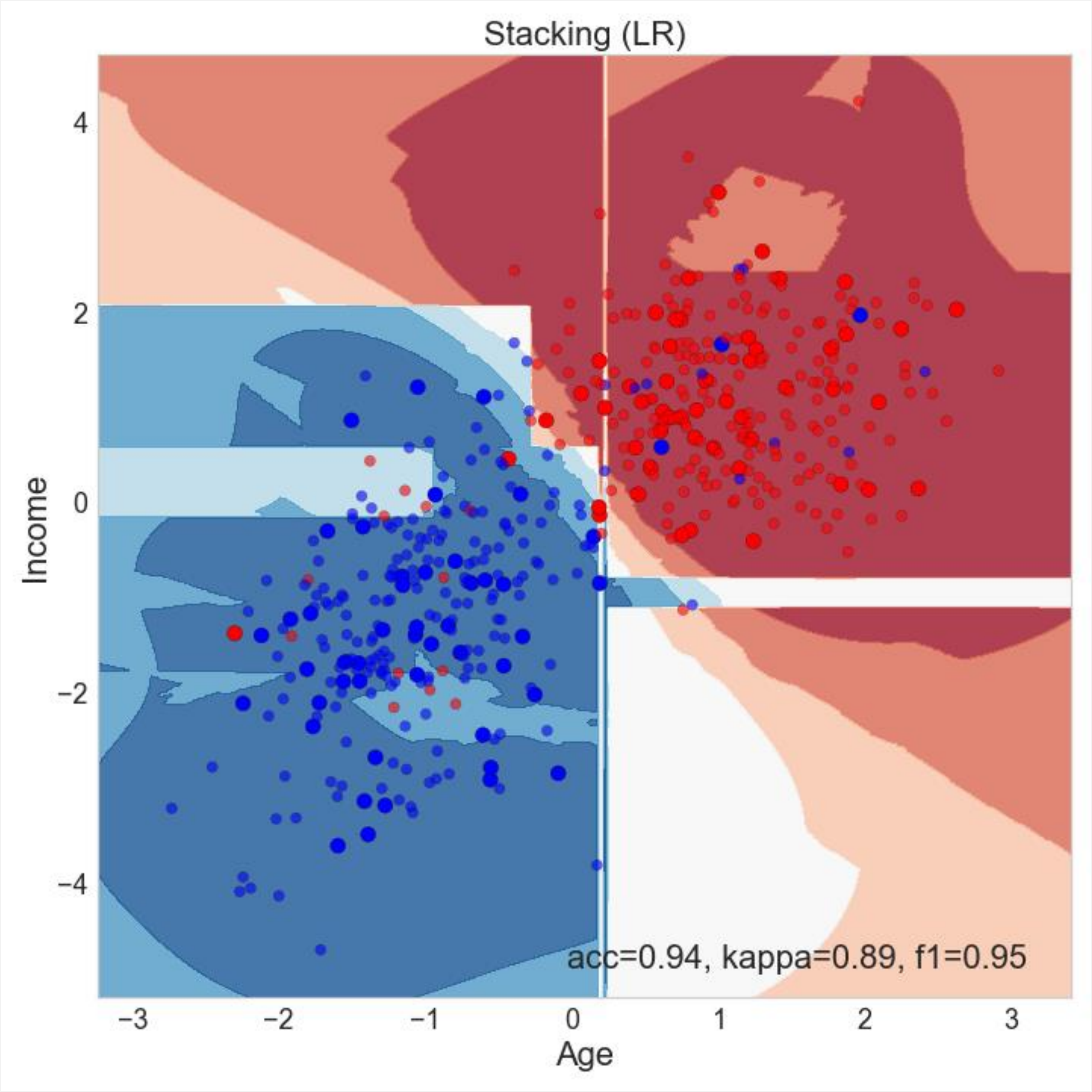
Example: Default Data



Example: Default Data



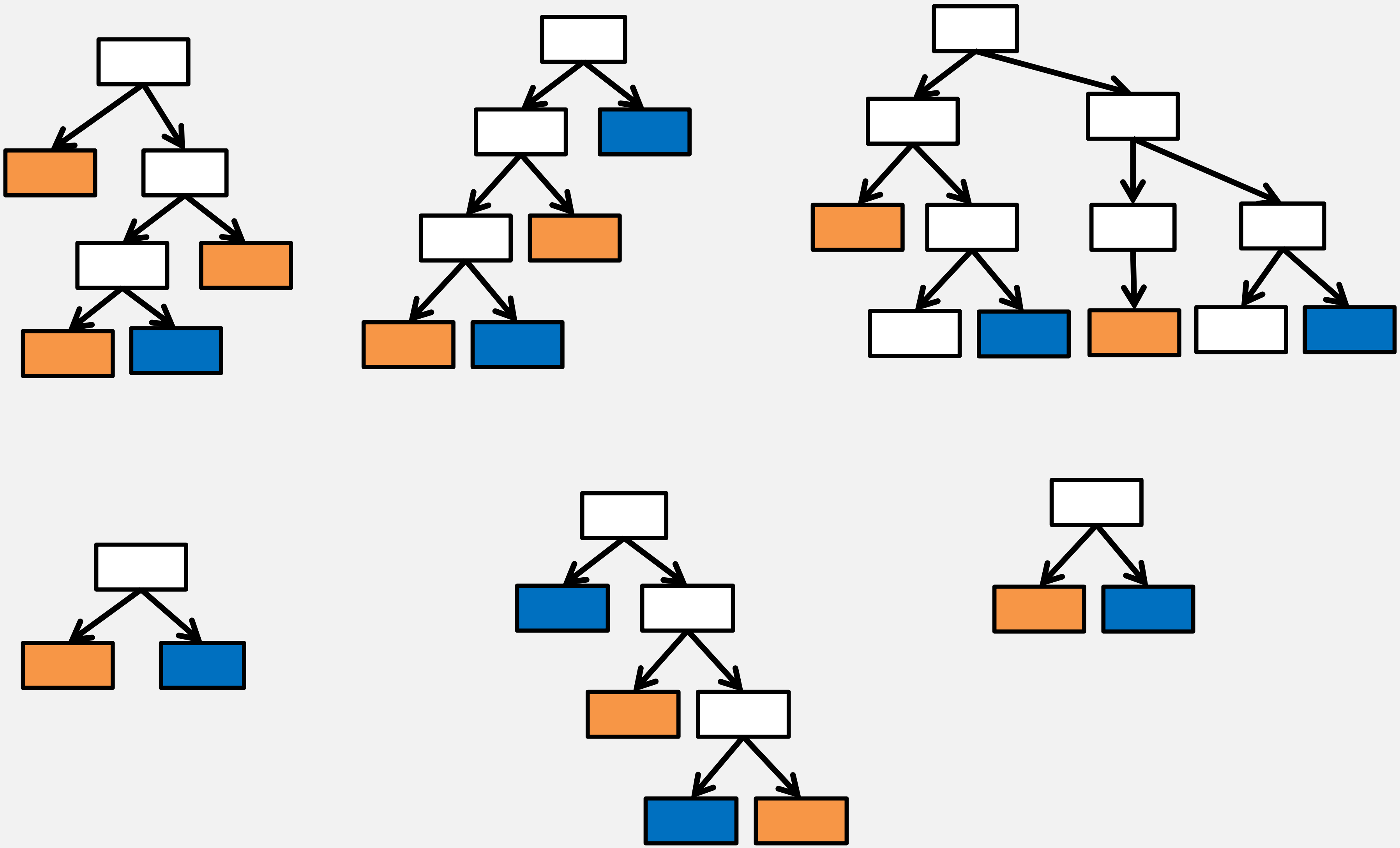
Example: Default Data



Example: Default Data

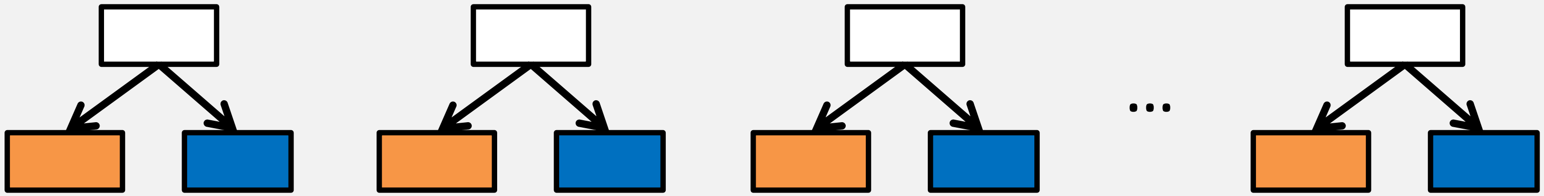
```
from sklearn.ensemble import BaggingClassifier

clf_bag = BaggingClassifier(
    DecisionTreeClassifier(max_depth=None, min_samples_split=2),
    n_estimators=100, max_samples=.10, max_features=0.5,random_state=0)
clf_bag.fit(X_train, y_train)
```

Random Forests



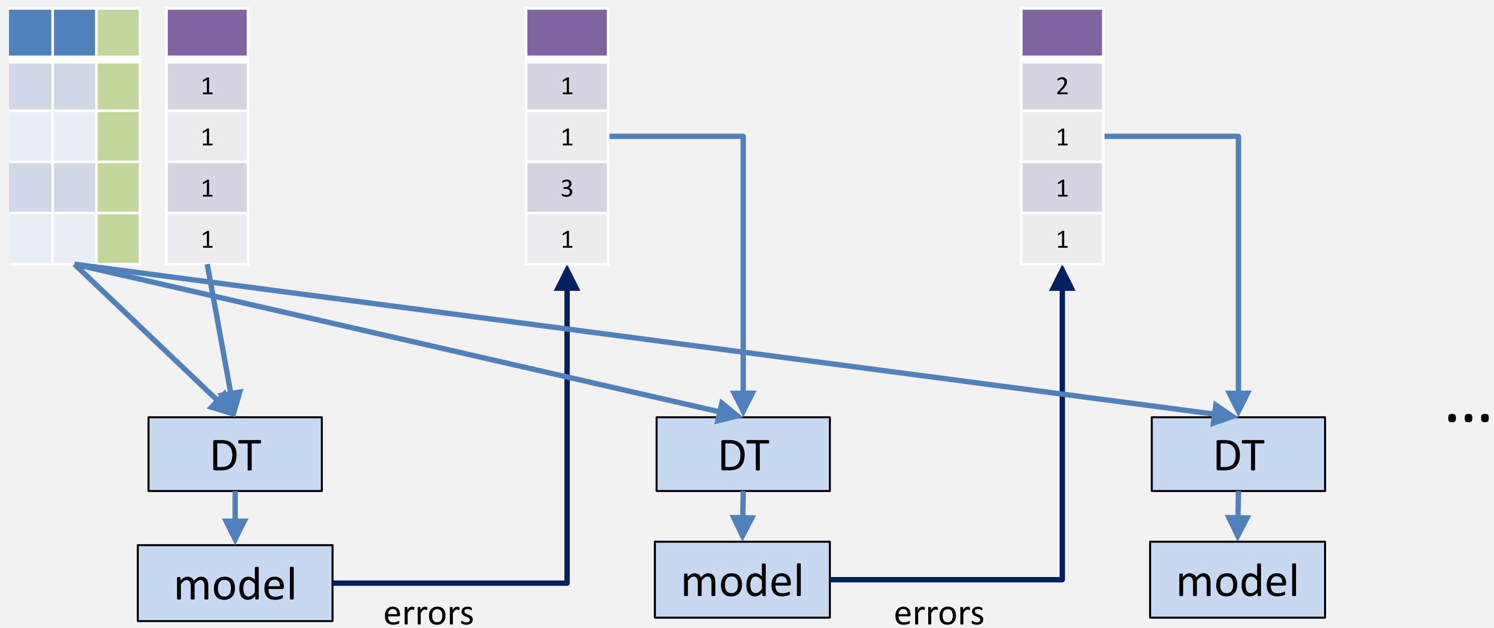
AdaBoost

- ***AdaBoost*** (Adaptive Boosting) was the first implementation of boosting
  - Won the 2003 Gödel Prize
- Like RFs, with three main differences:
  - Boosting
  - Stumps, not trees
  - Not all equal: stumps get different "say" in final classification

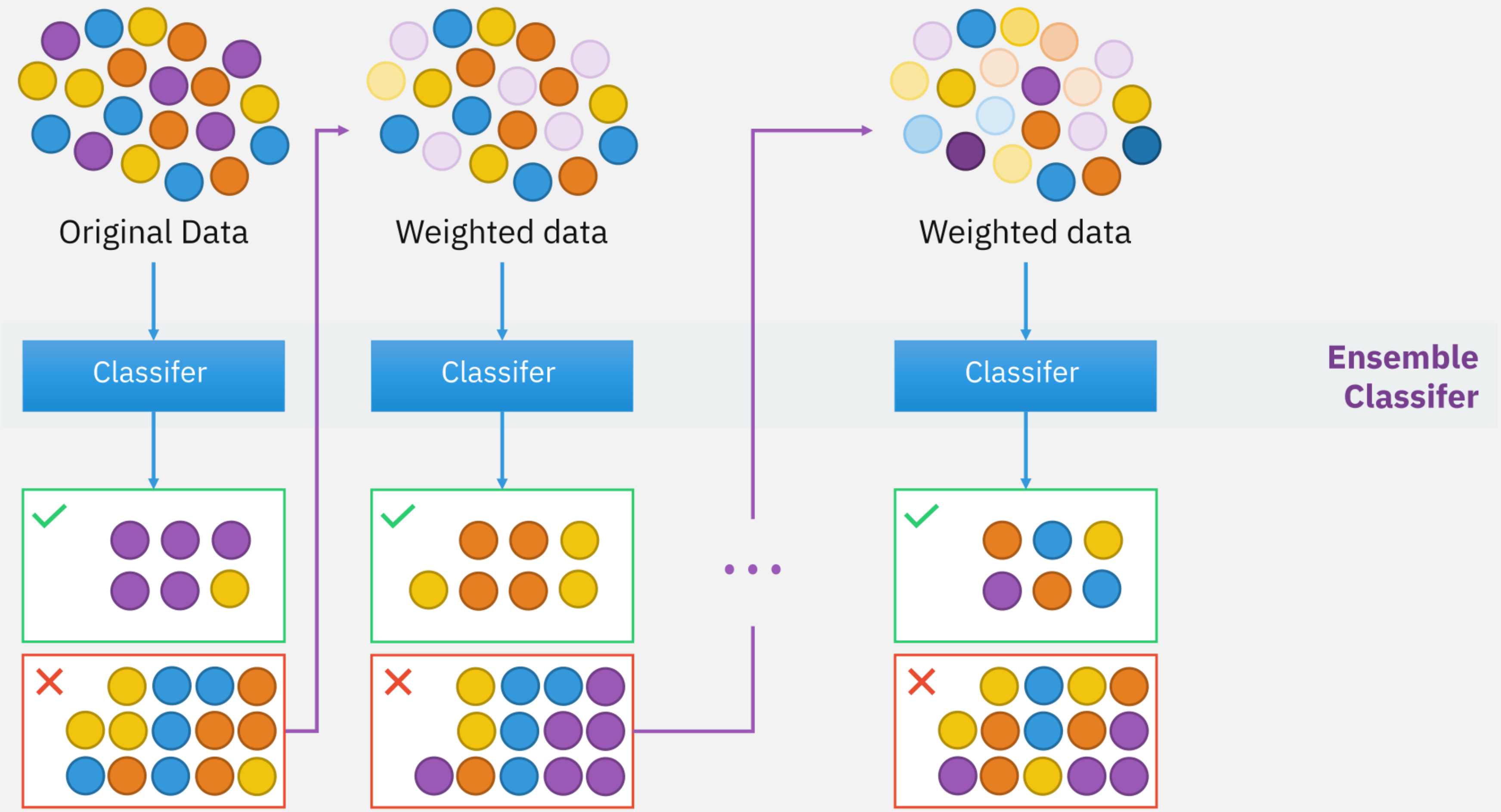


AdaBoost Algorithm

- Train a DT stump as normal*
  - *DTs now use Weighted Gini Index metric to account for weights
- Get list incorrectly-classified instances
- Increase weights for next DT
- Rinse and repeat for other DTs

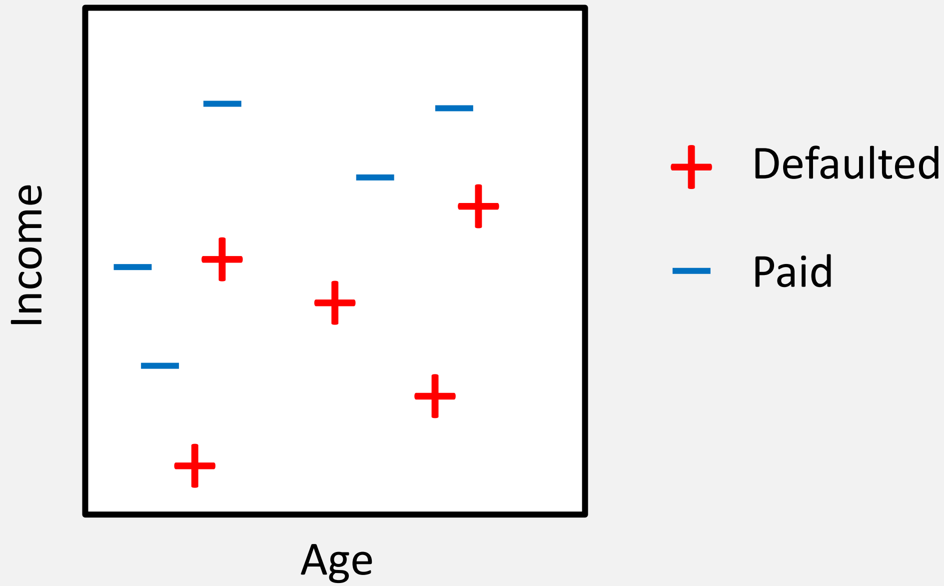


Visual Example (Wikipedia)

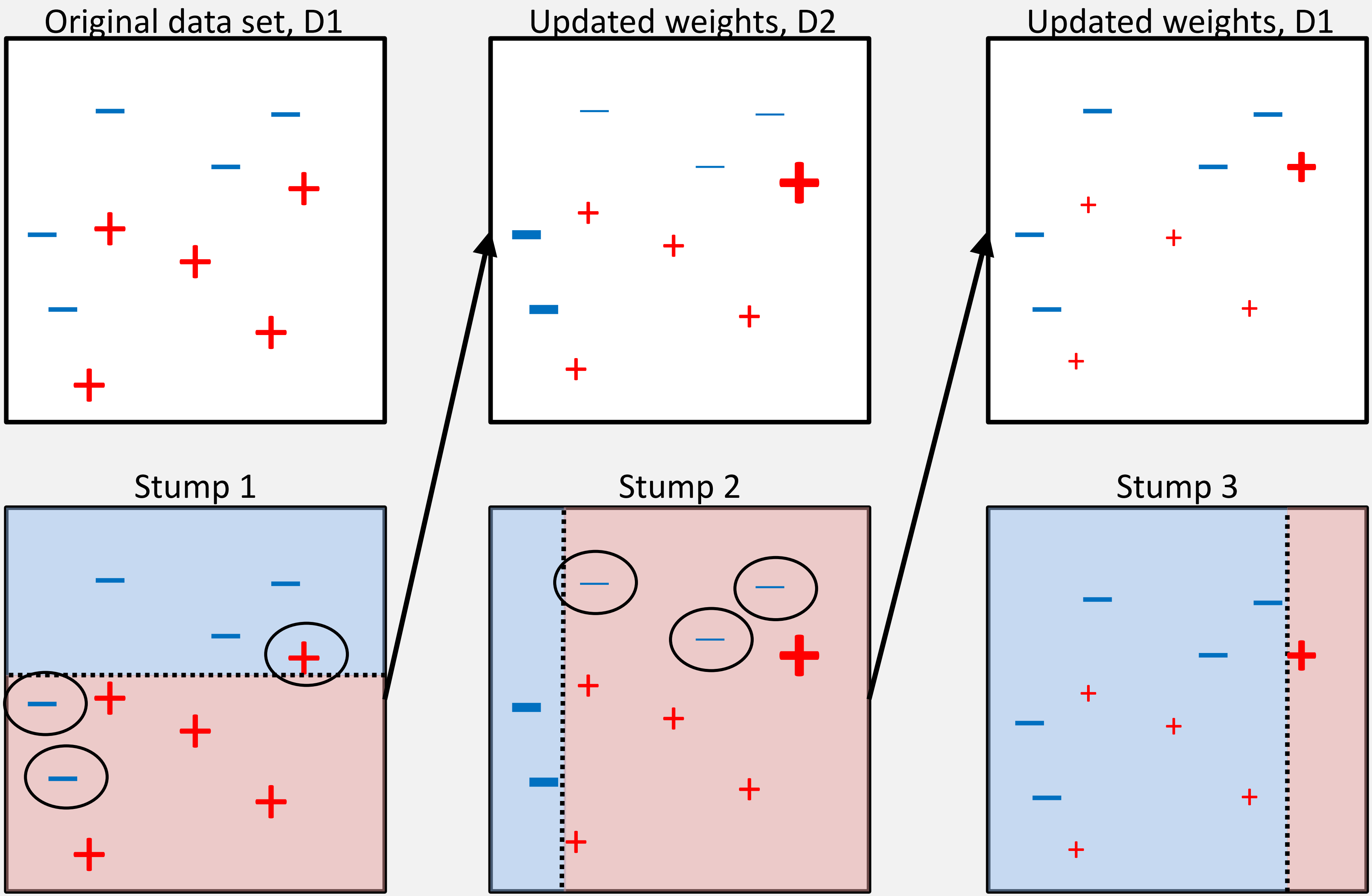


Visual Example (Textbook)

- Say we have a loan default dataset with two features, Age and Income



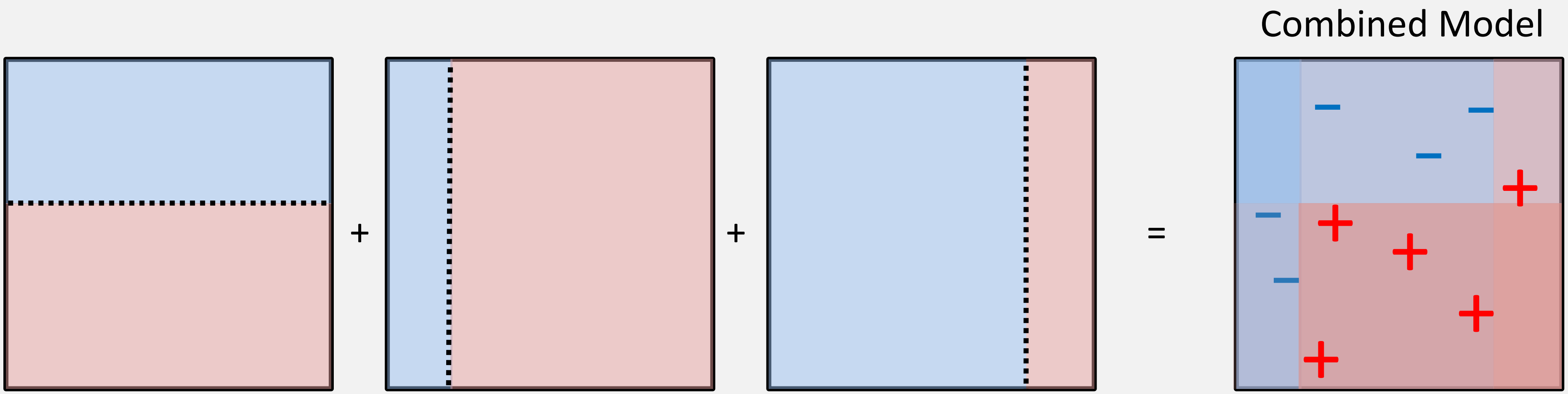
Example



Source: https://www.youtube.com/watch?v=ix6IvwbVpw0

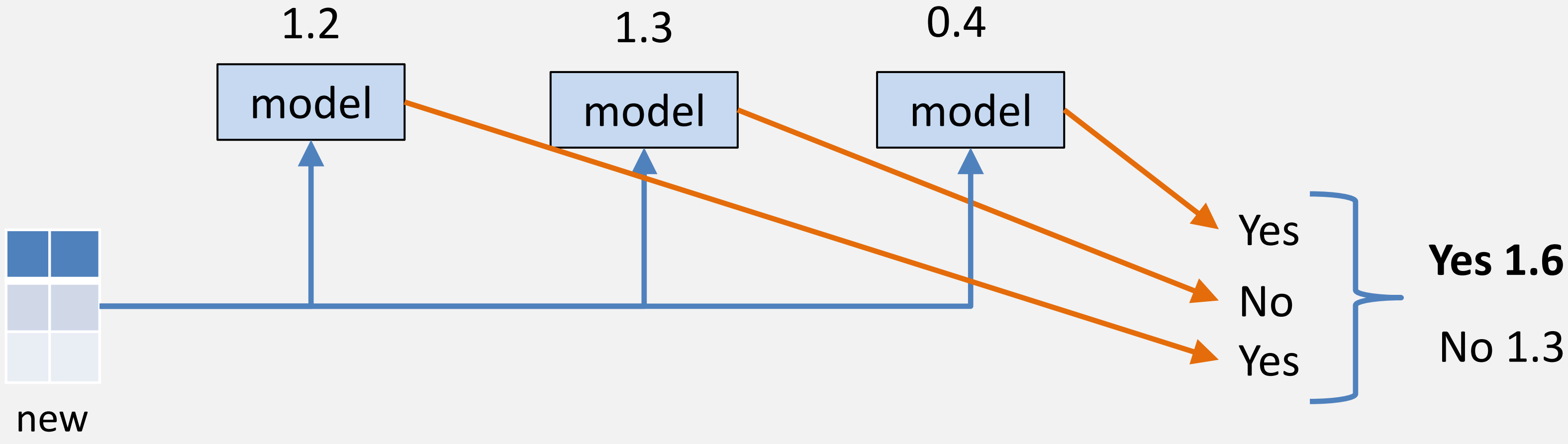
Example

- Each individual decision stump is very simple
- Final ensemble works very well!



Predictions

- Each DT's "*amount of say*" in final prediction is determined by how accurate that DT was on training data
  - (*Amount of Say* formula omitted here)
- To predict new, unlabeled data:
  - Each DT makes prediction
  - Prediction with highest Amount of Say wins



Example: Default Data

```python
from sklearn.ensemble import AdaBoostClassifier

clf_ada = AdaBoostClassifier(
    base_estimator=DecisionTreeClassifier(max_depth=1),
    n_estimators=100, random_state=0)
clf_ada.fit(X_train, y_train)
```

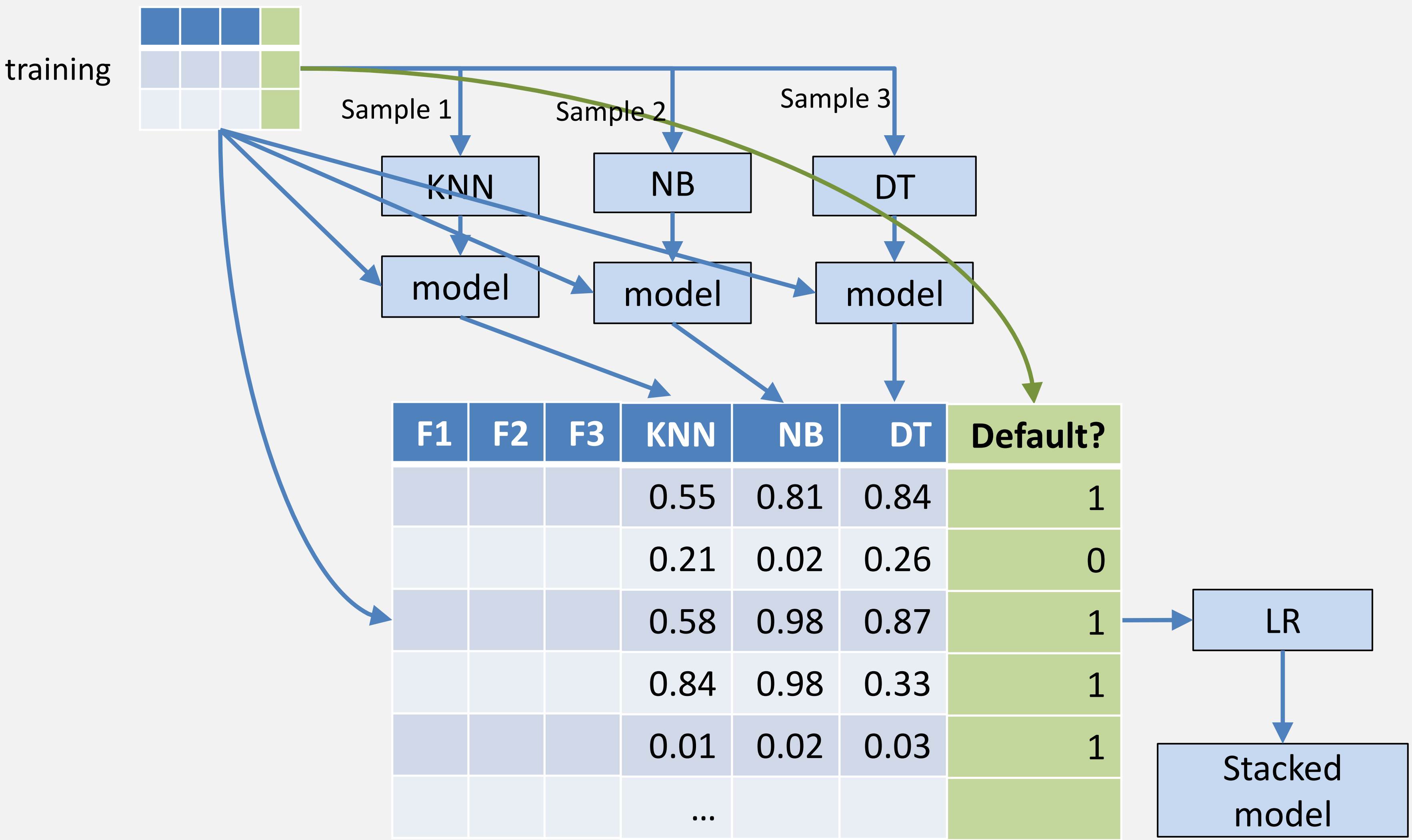
Example: Default Data

```
from xgboost import XGBClassifier
clf_xg = XGBClassifier(n_estimators=100, max_depth=3)
clf_xg.fit(X_train, y_train)
```

STACKING

Stacking (Super Learning)

- Like bagging, except uses predictions of each model as additional features for a new model



F1	F2	F3	KNN	NB	DT	Default?
			0.55	0.81	0.84	1
			0.21	0.02	0.26	0
			0.58	0.98	0.87	1
			0.84	0.98	0.33	1
			0.01	0.02	0.03	1
			…			

Example: Default Data

```
from mlxtend.classifier import StackingClassifier
from sklearn.linear_model import LogisticRegression

clf1 = DecisionTreeClassifier(max_depth=4)
clf2 = KNeighborsClassifier(n_neighbors=7)
clf3 = SVC(kernel='rbf', probability=True)

classifiers=[('DT', clf1), ('KNN', clf2), ('SVM', clf3)]

sclf = StackingClassifier(
    classifiers=classifiers, meta_classifier=LogisticRegression(),
    use_probas=True, average_probas=False)

clf1 = clf1.fit(X_train, y_train)
clf2 = clf2.fit(X_train, y_train)
clf3 = clf3.fit(X_train, y_train)
sclf = cclf.fit(X_train, y_train)
```

COMPARISON

Uncle Steve's Guide to Ensemble Techniques

Technique	Summary	Pros	Cons
All		✓ Improves accuracy ✓ More robust	× Reduces interpretability × Time consuming
Bagging (RF, ET)	*Build N models in parallel; combine their output*	✓ Fastest	
Boosting (Ada, XG, Cat, LGBM)	*Build N models sequentially, using the previous models' errors; combine their output*	✓ Higher accuracy	× Slower
Stacking	*Like bagging, but the model weights are learned via supervised learning*	✓ Highest accuracy	× Slowest

Example on Diabetes Dataset

	Dataset	Method	Time	Accuracy	Recall	Precision	F1	AUC	Rank
7	Diabetes	Adaboost	0.367171	0.772727	0.648148	0.686275	0.666667	0.744074	1
8	Diabetes	GBC	0.300580	0.753247	0.629630	0.653846	0.641509	0.724815	2
3	Diabetes	Voting	0.045212	0.746753	0.592593	0.653061	0.621359	0.711296	3
5	Diabetes	RF	0.377052	0.746753	0.592593	0.653061	0.621359	0.711296	4
4	Diabetes	Bagging	0.619293	0.746753	0.574074	0.659574	0.613861	0.707037	5
1	Diabetes	NB	0.002234	0.707792	0.648148	0.573770	0.608696	0.694074	6
9	Diabetes	Stacking	4.499238	0.733766	0.555556	0.638298	0.594059	0.692778	7
6	Diabetes	ExtraTrees	0.288043	0.727273	0.555556	0.625000	0.588235	0.687778	8
2	Diabetes	DT	0.006047	0.727273	0.500000	0.642857	0.562500	0.675000	9
0	Diabetes	LR	0.038450	0.714286	0.518519	0.608696	0.560000	0.669259	10

Example on German Credit Dataset

	Dataset	Method	Time	Accuracy	Recall	Precision	F1	AUC	Rank
8	GermanCredit	GBC	0.431021	0.760	0.857143	0.810811	0.833333	0.695238	1
6	GermanCredit	ExtraTrees	0.359332	0.745	0.857143	0.794702	0.824742	0.670238	2
5	GermanCredit	RF	0.408510	0.735	0.885714	0.770186	0.823920	0.634524	3
9	GermanCredit	Stacking	6.353294	0.725	0.828571	0.789116	0.808362	0.655952	4
4	GermanCredit	Bagging	1.137991	0.720	0.828571	0.783784	0.805556	0.647619	5
0	GermanCredit	LR	0.125056	0.700	0.792857	0.781690	0.787234	0.638095	6
3	GermanCredit	Voting	0.166601	0.705	0.771429	0.800000	0.785455	0.660714	7
7	GermanCredit	Adaboost	0.482291	0.695	0.778571	0.784173	0.781362	0.639286	8
2	GermanCredit	DT	0.011358	0.675	0.742857	0.781955	0.761905	0.629762	9
1	GermanCredit	NB	0.004378	0.685	0.714286	0.813008	0.760456	0.665476	10

	Dataset	Method	Time	Accuracy	Recall	Precision	F1	AUC	Rank
7	Adult	Adaboost	8.848156	0.871488	0.652423	0.777947	0.709677	0.796687	1
8	Adult	GBC	13.333880	0.872563	0.642857	0.788732	0.708363	0.794128	2
5	Adult	RF	8.524249	0.856902	0.636480	0.733824	0.681694	0.781637	3
4	Adult	Bagging	46.175066	0.854445	0.637755	0.724638	0.678426	0.780455	4
9	Adult	Stacking	93.588830	0.857362	0.605230	0.753773	0.671383	0.771270	5
6	Adult	ExtraTrees	10.342284	0.836634	0.616709	0.676224	0.645097	0.761540	6
2	Adult	DT	0.331754	0.814525	0.635204	0.610294	0.622500	0.753295	7
3	Adult	Voting	0.965517	0.809304	0.306760	0.756289	0.436479	0.637708	8
1	Adult	NB	0.089855	0.799324	0.317602	0.677551	0.432479	0.634837	9
0	Adult	LR	0.504623	0.799478	0.274235	0.719064	0.397045	0.620130	10

Rank on 12 Datasets

Method	count	mean	std	min	25%	50%	75%	max
Stacking	12.0	3.666667	2.708013	1.0	1.75	3.5	4.25	10.0
Bagging	12.0	4.083333	1.729862	1.0	3.00	4.0	5.00	8.0
GBC	12.0	4.333333	2.870962	1.0	2.00	4.0	5.75	9.0
RF	12.0	4.500000	2.430862	1.0	3.00	4.5	5.25	10.0
Adaboost	12.0	4.583333	3.175426	1.0	1.00	5.5	7.25	9.0
ExtraTrees	12.0	4.916667	2.644319	1.0	2.75	5.0	6.50	9.0
Voting	12.0	5.750000	2.340357	1.0	3.75	7.0	7.00	8.0
DT	12.0	7.333333	2.059715	3.0	6.00	7.5	9.00	10.0
NB	12.0	7.666667	3.025147	2.0	6.00	9.0	10.00	10.0
LR	12.0	8.166667	1.800673	5.0	6.75	8.5	10.00	10.0

Runtime on 12 Datasets

Method	count	mean	std	min	25%	50%	75%	max
NB	12.0	0.023328	0.031528	0.001997	0.004321	0.009829	0.023199	0.089855
DT	12.0	0.228829	0.414707	0.003030	0.010509	0.041785	0.132864	1.282202
ExtraTrees	12.0	2.084052	2.888931	0.208411	0.355067	0.777813	3.060318	10.342284
LR	12.0	2.714115	6.718642	0.038450	0.168958	0.441879	1.223718	23.816978
Voting	12.0	2.906445	6.829440	0.045212	0.196348	0.657037	1.546212	24.277507
RF	12.0	4.127439	7.658385	0.284079	0.407788	1.305802	3.204870	27.195963
Adaboost	12.0	5.404998	9.917278	0.294612	0.464008	1.154748	4.589717	34.736051
GBC	12.0	11.356762	25.335455	0.170561	0.405727	1.557293	6.126897	88.876732
Bagging	12.0	27.121253	48.869182	0.371966	1.036678	3.529258	16.347215	140.105755
Stacking	12.0	61.200335	97.879150	4.492594	6.205815	19.862836	48.011568	320.393715

Boosting

- ***Boosting:*** trains new models to correct mistakes of previous models
- Variants:
  - Adaptive Boosting (e.g., AdaBoost): Original, but not the best anymore
  - Gradient Boosting (e.g., XGBoost): Very popular! Great results