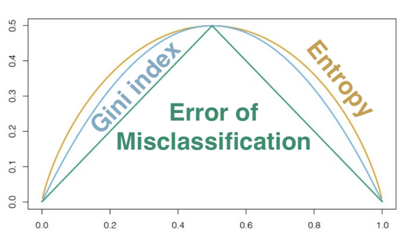
Decision Trees

Python

```python
from sklearn import tree
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X_train, y_train)
clf.predict(X_test)
clf.predict_proba(X_test)
tree.plot_tree(clf)
```

Metrics used for splitting

Node impurity measures for two-class classification, as a function of the proportion p in class 2. Cross-entropy has been scaled to pass through (0.5, 0.5).



Algorithm

Algorithm 8.1 *Building a Regression Tree*

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.
2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.
3. Use K-fold cross-validation to choose α. That is, divide the training observations into K folds. For each $k = 1, \dots, K$:
   (a) Repeat Steps 1 and 2 on all but the kth fold of the training data.
   (b) Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.
   Average the results for each value of α, and pick α to minimize the average error.
4. Return the subtree from Step 2 that corresponds to the chosen value of α.

Disadvantages of trees

Instability of trees

One major problem with trees is their high variance. Often a small change in the data can result in a very different series of splits, making interpretation somewhat precarious. The major reason for this instability is the hierarchical nature of the process: the effect of an error in the top split is propagated down to all of the splits below it. One can alleviate this to some degree by trying to use a more stable split criterion, but the inherent instability is not removed. It is the price to be paid for estimating a simple, tree-based structure from the data. Bagging (Section 8.7) averages many trees to reduce this variance.

Lack of smoothness

Another limitation of trees is the lack of smoothness of the prediction surface, as can be seen in the bottom right panel of Figure 9.2. In classification with 0/1 loss, this doesn't hurt much, since bias in estimation of the class probabilities has a limited effect. However, this can degrade performance in the regression setting, where we would normally expect the underlying function to be smooth.

Binary vs. Multiway Splits

Rather than splitting each node into just two groups at each stage (as above), we might consider multiway splits into more than two groups. While this can sometimes be useful, it is not a good general strategy. The problem is that multiway splits fragment the data too quickly, leaving insufficient data at the next level down. Hence we would want to use such splits only when needed. Since multiway splits can be achieved by a series of binary splits, the latter are preferred.

Comparison with other methods

	Trees	SVM	Neural Nets	kNN / Kernels
Natural handling of mixed type data	✓	✗	✗	✗
Handling of missing values	✓	✗	✗	✓
Robustness to outliers	✓	✗	✗	✓
Insensitive to monotone transforms	✓	✗	✗	✗
Computation scalability	✓	✗	✗	✗
Ability to deal with irrelevant inputs	✓	✗	✗	✗
Ability to extract linear combinations	✗	✓	✓	●
Interpretability	●	✗	✗	✗
Predictive power	✗	✓	✓	✓

Gradient Boosting

Python

```python
# XGBOOST EXAMPLE
import xgboost as xgb

dtrain = xgb.DMatrix('train.csv')
dtest = xgb.DMatrix('test.csv')
param = {'max_depth': 2, 'eta': 1}
num_round = 2
bst = xgb.train(param, dtrain,
num_round)

preds = bst.predict(dtest)


# CATBOOST EXAMPLE
from catboost import
CatBoostClassifier
from catboost import Pool

train_pool = Pool(
   data=X_train,
   label=y_train,
  #cat_features=cat_features
)

validation_pool = Pool(
   data=X_test,
   label=y_test,
  #cat_features=cat_features
)

model = CatBoostClassifier(
   iterations=50,
   learning_rate=0.01,
   custom_loss=['AUC', 'Accuracy']
)

model.fit(
   train_pool,
   eval_set=validation_pool,
   verbose=False,
   plot=True
)
```

Algorithm

Algorithm 8.2 *Boosting for Regression Trees*

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.
2. For $b = 1, 2, \dots, B$, repeat:
   (a) Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r).
   (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \qquad (8.10)$$
   (c) Update the residuals,
   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \qquad (8.11)$$
3. Output the boosted model,
   $$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \qquad (8.12)$$

Advantages of GBM

- Robust to noise
- Resilient to correlated variables
- Monotonic transformations have little/no effect
- Regularisation to avoid overfitting
- Early stopping
- Parallelised tree building
- All three algorithms can be trained on GPU

Categorical variables

- (Internal in libraries)
- Label encoding
- One-hot encoding
- Count encoding
- Percent encoding
- Target encoding (see below)

Target encoding

- Overfits easily
- Catboost uses in a way
- Impl.: category_encoders
- Parameters
  - one_hot_max_size
  - te_smothing,
  - te_min_samples
- Calculate on train/cv_train
- Treatment of new cats
- Missing values

$$\mu = \begin{cases} \dfrac{n * \bar{x} + m * w}{n + m}, & n \geq z \\ w, & n < z \end{cases}$$

μ : Mean of y per category (target encoded)
n : The number of observations in category
$\bar{x}$: The estimated mean
m: The "weight" you want to assign to the overall mear
w: The overall mean
z : Minimum number of records to target encode

XGBoost Hyperparameters

General parameters

- `max_depth`
- `learning_rate` (shrinkage, eta)
- `max_delta_step` (additional cap on learning rate, needed in case of highly imbalanced classes)
- `n_estimators` (number of boosting rounds)
- `booster` (usually trees)
- `scale_pos_weight` (for rebalancing weights of positive or negative samples in binary classification)
- `base_score` (by default predicting 0.5 for all observations)
- `seed/random_state`
- `missing` (for treating other values as missing)
- `objective` (callable(y, y_hat))

Random subsampling

- `subsample` (percentage of rows)
- `colsample_bytree` (percentage of columns when constructing each tree)
- `colsample_bylevel` (percentage of columns when building each level of three tree)

Regularisation parameters

- `reg_alpha` (L1/lasso regularisation)
- `reg_labmda` (L2/ridge regularisation)
- `gamma` (L0 regularisation, dependent on number of leaves)
- `min_child_weight` (min sum of hessians in a child)
- `min_child_weight` (for binary classification, sum of second order gradients)

Hyperparameter optimisation

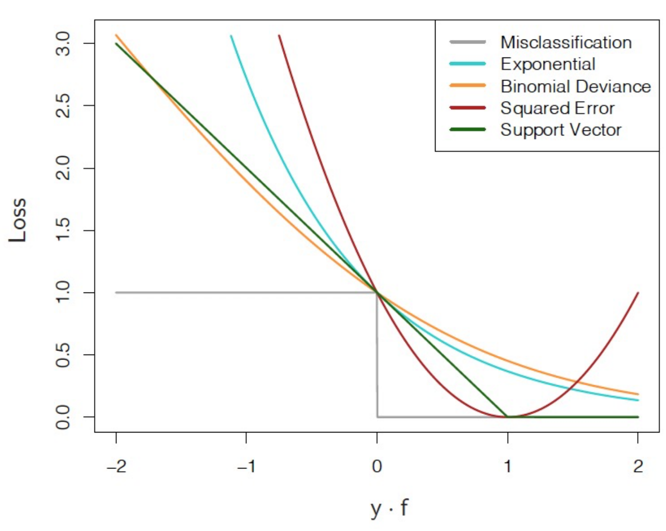
Methods

- Grid search (e.g. `GridSearchCV`)
- Random search (e.g. `sklearn.model_selection.RandomizedSearchCV`)
- Bayesian Hyperparameter optimisation, using (e.g. `hyperopt`)

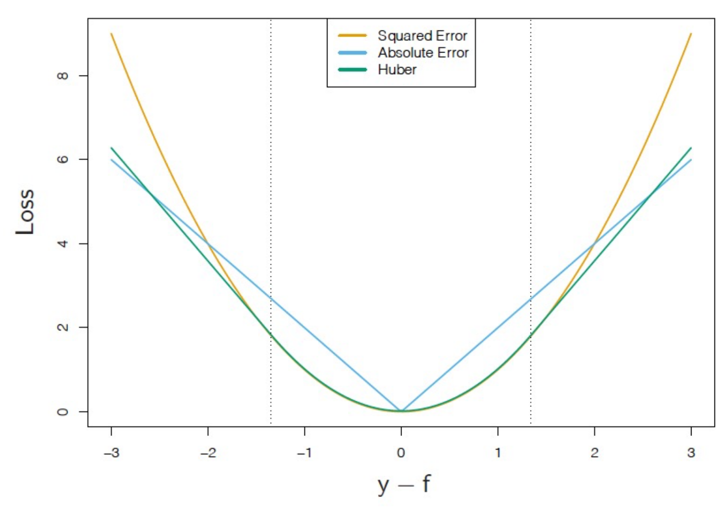
Steps for optimization using hyperopt

- Defining the objective function. This includes hyperparameter-dependent transformations and cross-validating results.
- Defining the hyperparameter search space, i.e. the distributions to sample from. E.g.: `hp.quniform('te_smoothing', 1, 10, 1)`
- Running the optimisation using `fmin` over a number iterations

Loss functions for binary classification



Loss functions for regression



Author: Vlad Corduneanu, https://www.linkedin.com/in/corduneanu/