

ML_Chap.3Notes

Summary/Review - Week1_Logistic Regression (2021.12.28)

Classification Problems

The two main types of supervised learning models are:

- Regression models, which predict a continuous outcome
- Classification models, which predict a categorical outcome.

The most common models used in supervised learning are:

- Logistic Regression
- K-Nearest Neighbors
- Support Vector Machines
- Decision Tree
- Neural Networks
- Random Forests
- Boosting
- Ensemble Models

With the exception of logistic regression, these models are commonly used for both regression and classification.

Logistic regression is most common for dichotomous and nominal dependent variables.

Logistic Regression

Logistic regression is a type of regression that models the probability of a certain class occurring given other independent variables. It uses a logistic or logit function to model a dependent variable. It is a very common predictive model because of its high interpretability.

Classification Error Metrics

A confusion matrix tabulates true positives, false negatives, false positives and true negatives.

Confusion Matrix

	Predicted Positive	Predicted Negative	
Actual Positive	True Positive (TP)	False Negative (FN)	Type II Error
Actual Negative	False Positive (FP)	True Negative (TN)	
Type I Error			

Remember that the false positive rate is also known as a **type I error**.

The false negatives are also known as a **type II error**.

Accuracy is defined as the ratio of true positives and true negatives divided by the total number of observations. It is a measure related to predicting correctly positive and negative instances.

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

Recall or sensitivity identifies the ratio of true positives divided by the total number of actual positives.

It quantifies the percentage of positive instances correctly identified.

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

Precision is the ratio of true positive divided by total of predicted positives. The closer this value is to 1.0, the better job this model does at identifying only positive instances.

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)
Precision = $\frac{TP}{TP + FP}$		

Specificity is the ratio of true negatives divided by the total number of actual negatives. The closer this value is to 1.0, the better job this model does at avoiding false alarms.

Error Measurements

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

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

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

$$\text{Specificity} = \frac{TN}{FP + TN}$$

$$F1 = 2 \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

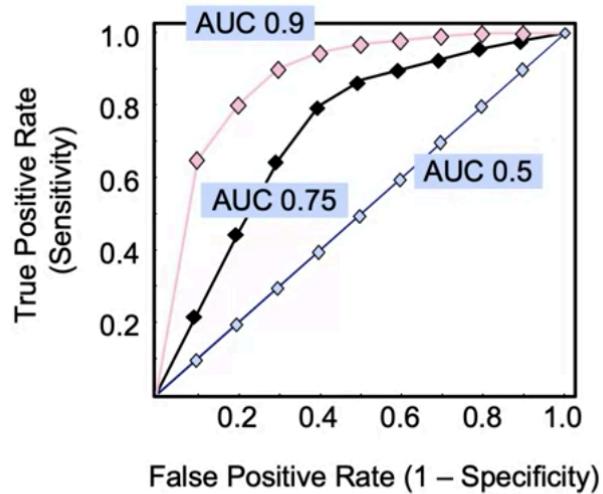
	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)



IBM

The receiver operating characteristic (**ROC**) plots the true positive rate (sensitivity) of a model vs. its false positive rate (1-sensitivity).

Receiver Operating Characteristic (ROC)

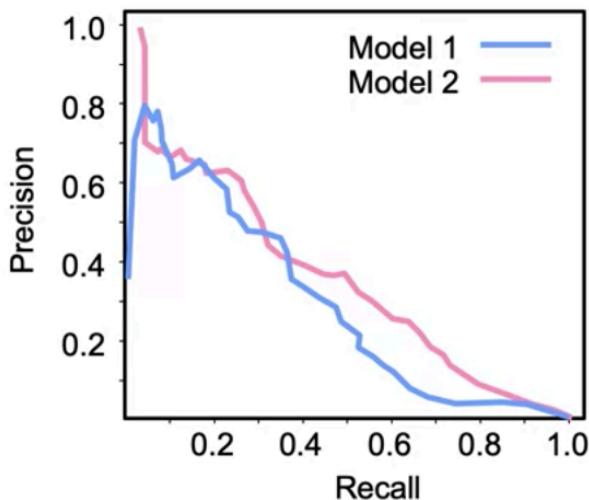


Measures total area under ROC curve

The area under the curve of a ROC plot is a very common method of selecting a classification methods.

The [precision-recall curve](#) measures the trade-off between precision and recall.

Precision-Recall Curve



Measures trade-off between precision and recall

The [ROC curve](#) generally works better for data with [balanced classes](#), while the [precision-recall curve](#) generally works better for data with [unbalanced classes](#).

Multiple Class Error Metrics

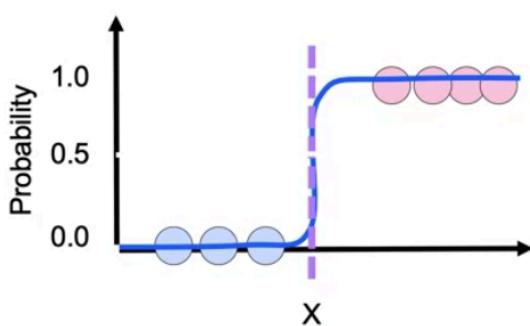
Most **multi-class error metrics** are similar to binary versions, they just expand elements as a sum.

$$\text{Accuracy} = \frac{\text{TP1} + \text{TP2} + \text{TP3}}{\text{Total}}$$

	Predicted Class 1	Predicted Class 2	Predicted Class 3
Actual Class 1	TP1		
Actual Class 2		TP2	
Actual Class 3			TP3

Overview of Classifier Characteristics

$$y_{\beta}(x) = \frac{1}{1+e^{-(\beta_0 + \beta_1 x + \varepsilon)}}$$



- For logistic regression, model is just parameters.
- Fitting can be slow — must find best parameters.
- Prediction is fast — calculate expected value.
- Decision boundary is simple, less flexible.

Week2

K Nearest Neighbor Methods for Classification (2021.12.29)

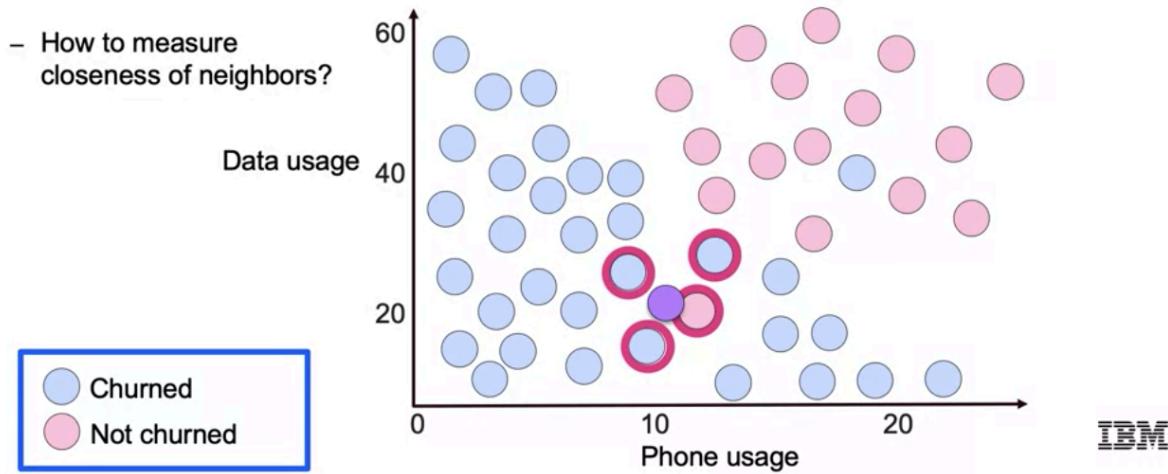
K nearest neighbor methods are useful for classification.

The **elbow method** is frequently used to identify a model with low K and low error rate.

These methods are popular due to their easy computation and interpretability, although it might take time scoring new observations, it lacks estimators, and might not be suited for large data sets.

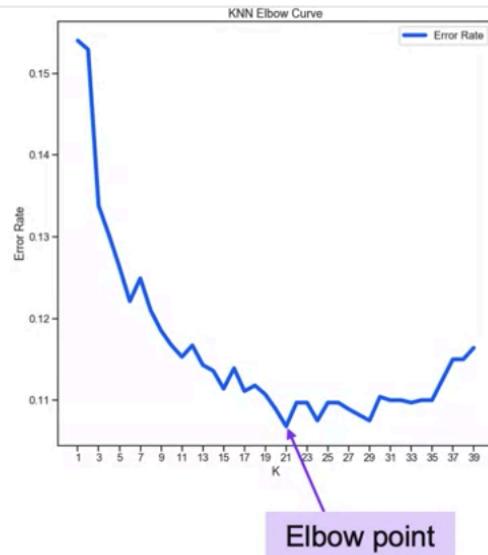
What is Needed to Select a KNN Model?

- Correct value for 'K'
- How to measure closeness of neighbors?

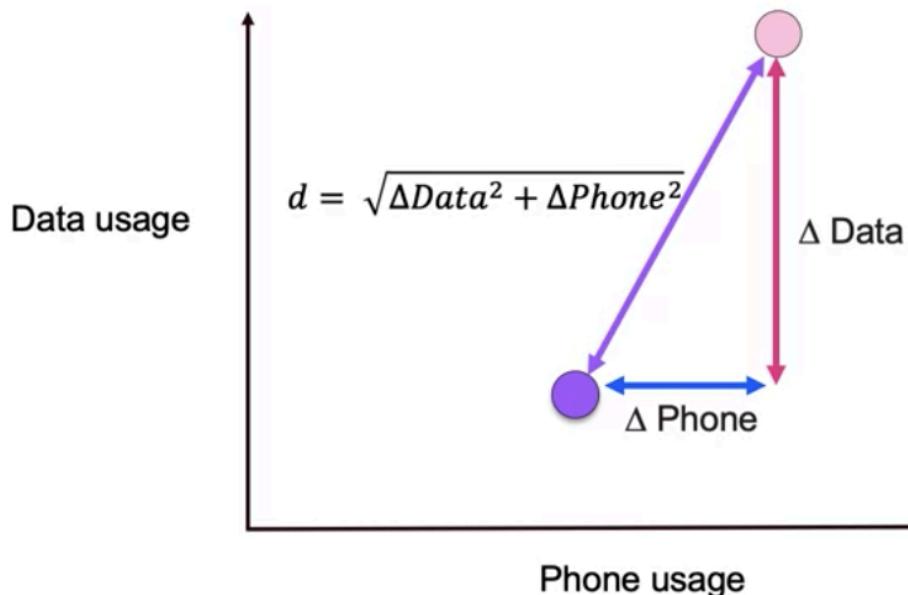


Choosing the right value for K

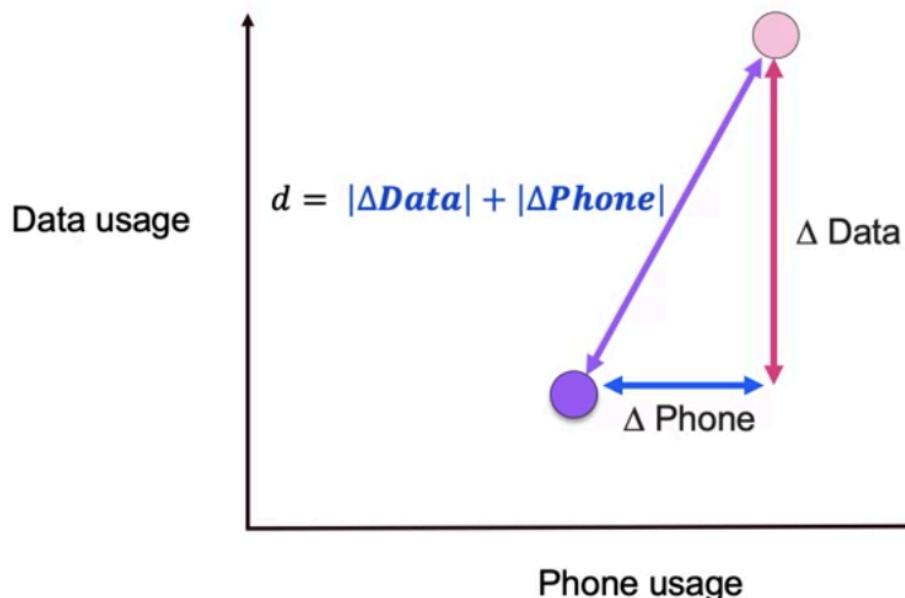
- KNN does not provide a 'correct' K
- The right value depends on which error metric is most important
- A common approach is to use an 'elbow method' approach
- This emphasizes kinks in a curve of the error rate as a function of K
- Beyond this point, the rate of improvement slows or stops



Euclidean Distance (L2)



Manhattan Distance (L1)



KNN Overview

Pros:

- Simple to implement (does not require estimation)
- Adapts well as new training data
- Easy to interpret

Cons:

- Slow to predict because many distance calculations
- Does not generate insight into data generating process (no model)
- Can require lots of memory if data set is large (or as it grows)
- When there are many predictors, KNN accuracy can break down due to curse of dimensionality

Linear Regression

- Fitting involves minimizing cost function (slow)
- Model has few parameters (memory efficient)
- Prediction involves calculation (fast)

K Nearest Neighbors

- Fitting involves storing training data (fast)
- Model has many parameters (memory intensive)
- Prediction involves finding closest neighbors (slow)

Code

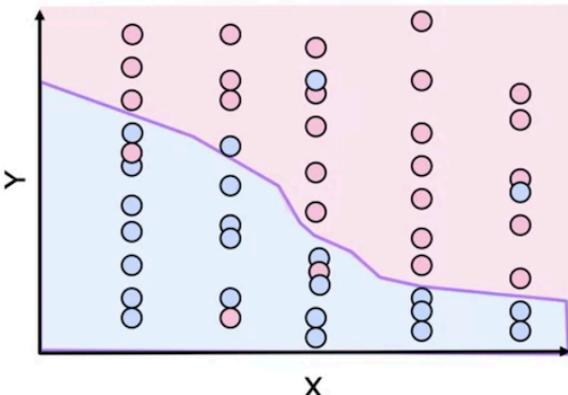
```
# Import the class containing the classification method
from sklearn.neighbors import KNeighborsClassifier

# Create an instance of the class
KNN = KNeighborsClassifier(n_neighbors=3)

# Fit the instance on the data and then predict the expected value
KNN = KNN.fit(X_train, y_train)
y_predict = KNN.predict(X_test)

Regression can be done with KNeighborsRegressor.
```

Overview of Classifier Characteristics



- For K-Nearest Neighbors, training data is the model.
- Fitting is fast — just store data.
- Prediction can be slow — lots of distances to measure.
- Decision boundary is flexible.

SVM (2021.12.30)

The main idea behind support vector machines is to **find a hyperplane** that separates classes by determining **decision boundaries** that maximize the distance between classes.

When comparing **logistic regression and SVMs**, one of the main differences is that the **cost function** for logistic regression has a cost function that decreases to zero, but rarely reaches zero. **SVMs use the Hinge Loss function as a cost function to penalize misclassification.** This tends to lead to better accuracy at the cost of having less sensitivity on the predicted probabilities.

Regularization can help SVMs generalize better with future data.

By using **gaussian kernels**, you transform your data space vectors into a different coordinate system, and may have better chances of finding a hyperplane that classifies well your data. SVMs with RBFs Kernels are **slow** to train with data sets that are large or have many features.

SVMs with Kernels: The Syntax

Code

```
# Import the class containing the classification method
from sklearn.svm import SVC

# Create an instance of the class
rbfSVC = SVC(kernel='rbf', gamma=1.0, C=10.0) "C" is penalty associated
# Fit the instance on the data and then predict the expected value
rbfSVC = rbfSVC.fit((X_train, y_train)
y_predict = rbfSVC.predict(X_test)

Tune kernel and associated parameters with cross-validation.
```

Faster Kernel Transformations: The Syntax

Code

```
# Import the class containing the classification method
from sklearn.kernel_approximation import Nystroem

# Create an instance of the class
NystroemSVC = Nystroem (kernel='rbf', gamma=1.0,
n_components=100) n_components is
# Fit the instance on the data and transform
X_train = NystroemSVC.fit_transform(X_train)
X_test = NystroemSVC.transform(X_test)

Tune kernel and associated parameters with cross-validation.
```

Machine Learning Workflow

Features

Data

Model Choice

- | | | |
|------------------------|-----------------------|---|
| - Many (~10K Features) | - Small (1K rows) | - Simple, Logistic or LinearSVC |
| - Few (<100 Features) | - Medium (~10k rows) | - SVC with RBF |
| - Few (<100 Features) | - Many (>100K Points) | - Add features, Logistic,
LinearSVC, or Kernel Approx. |

Wk3_Decision Tress (2021.12.31)

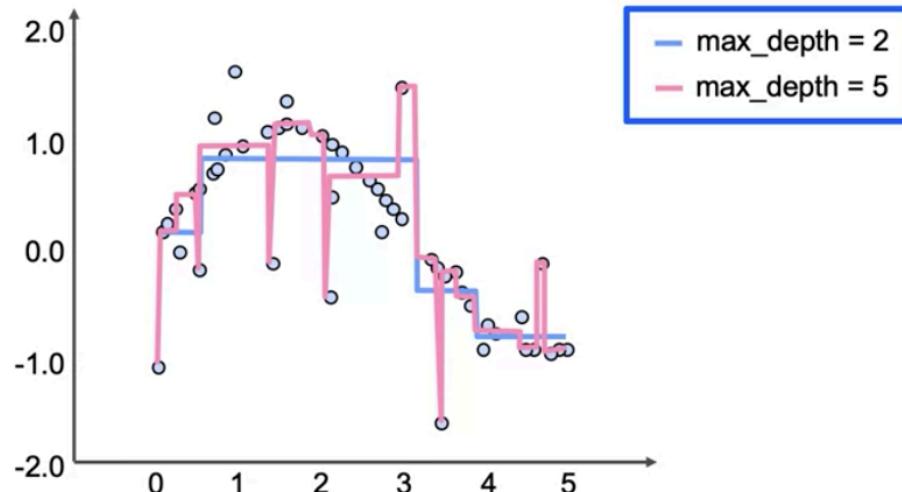
Decision trees split your data using impurity measures.

They are a **greedy algorithm** and are **not based on statistical assumptions**.

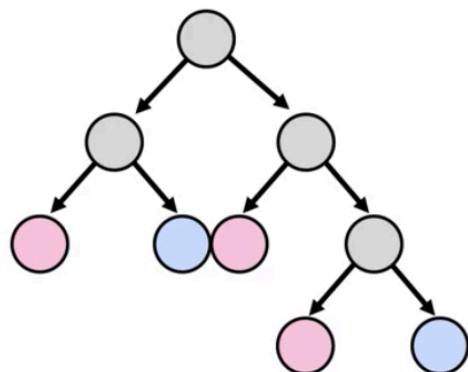
The most common splitting impurity measures are **Entropy and Gini index**.

Decision trees **tend to overfit** and to be very sensitive to different data.

Trees Predict Continuous Values



How Long to Keep Splitting?

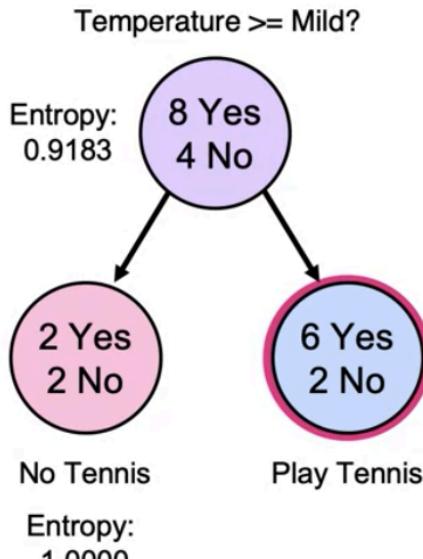


Until:

- Leaf node(s) are **pure** (only **one class** remains).
- A **maximum depth** is reached.
- A **performance metric** is achieved.



Splitting Based on Entropy



- Entropy Equation

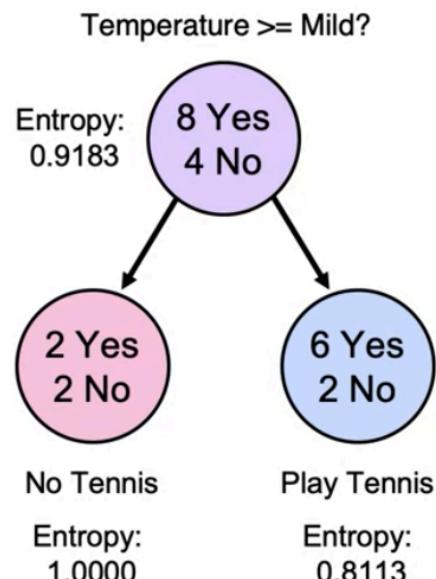
$$H(t) = - \sum_{i=1}^n p(i|t) \log_2[p(i|t)]$$

- Entropy Right Side

$$- \frac{6}{8} \log_2(\frac{6}{8}) - \frac{2}{8} \log_2(\frac{2}{8})$$

$$= 0.8113$$

Splitting Based on Entropy



- Classification Error Equation

$$E(t) = 1 - \max_i [p(i|t)]$$

- Entropy Change

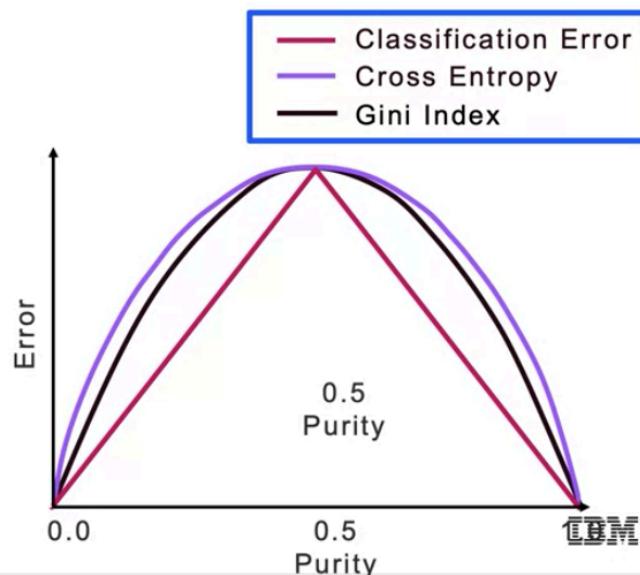
$$0.9183 - \frac{4}{12} * 1.0000 - \frac{8}{12} * 0.8113$$

$$= 0.0441$$

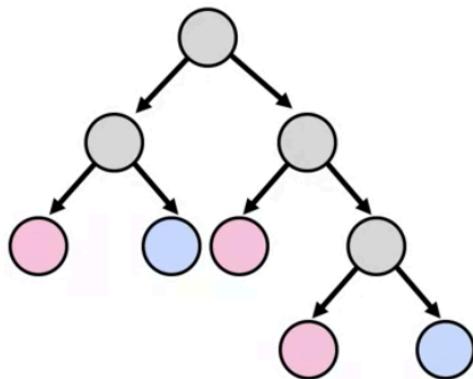
The Gini Index

- In practice, **Gini index** often used for splitting.
- Function is similar to entropy — has bulge.
- Does not contain logarithm.

$$G(t) = 1 - \sum_{i=1}^n p(i|t)^2$$



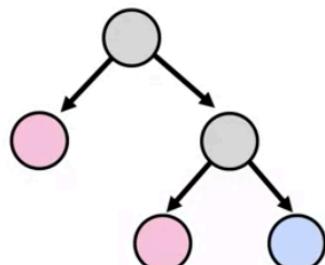
Decision Trees are High Variance



- Problem: decision trees tend to overfit.
- Small changes in data greatly affect prediction - high variance.
- Solution: Prune trees.



Pruning Decision Trees



- How to decide what leaves to prune?
- Solution: prune based on classification error threshold.

$$E(t) = 1 - \max_i[p(i|t)]$$

Cross validation and pruning sometimes help with some of this.

Great **advantages** of decision trees are that they are really easy to interpret and require no data preprocessing.

Code

```
# Import the class containing the classification method
from sklearn.tree import DecisionTreeClassifier

# Create an instance of the class
DTC = DecisionTreeClassifier (criterion='Gini',
max_features=10, max_depth=5)

# Fit the instance on the data and then predict the expected
# value
DTC = DTC.fit(X_train, y_train)
y_predict = DTC.predict(X_test)

Tune parameters with cross-validation. Use
DecisionTreeRegressor for regression.
```

tree parameters



Stratified-Shuffle-Split Method

- Use StratifiedShuffleSplit to split data into train and test sets that are stratified by wine quality. If possible, preserve the indices of the split for question 5 below.
- Check the percent composition of each quality level for both the train and test data sets.

```
### BEGIN SOLUTION
# All data columns except for color
feature_cols = [x for x in data.columns if x not in 'color']

from sklearn.model_selection import StratifiedShuffleSplit

# Split the data into two parts with 1000 points in the test data
# This creates a generator
strat_shuff_split = StratifiedShuffleSplit(n_splits=1, test_size=1000, random_state=42)

# Get the index values from the generator
train_idx, test_idx = next(strat_shuff_split.split(data[feature_cols], data['color']))

# Create the data sets
X_train = data.loc[train_idx, feature_cols]
y_train = data.loc[train_idx, 'color']

X_test = data.loc[test_idx, feature_cols]
y_test = data.loc[test_idx, 'color']
```

Now check the percent composition of each quality level in the train and test data sets. The data set is mostly white wine, as can be seen below.

```
y_train.value_counts(normalize=True).sort_index()

0    0.753866
1    0.246134
Name: color, dtype: float64
```

remarks

n_splits=1 (only split once on 'y' (wine color) =
white / red, two outcomes)
test_size=1000 (rows, before used percentage on
train_test_split)
next() – built-in-function in python, 迭代
data.loc[train_idx, feature_cols]

GridSearchCV to prune tree

```

: ### BEGIN SOLUTION
from sklearn.model_selection import GridSearchCV

param_grid = {'max_depth':range(1, dt.tree_.max_depth+1, 2),
              'max_features': range(1, len(dt.feature_importances_)+1)}

GR = GridSearchCV(DecisionTreeClassifier(random_state=42),
                  param_grid=param_grid,
                  scoring='accuracy',
                  n_jobs=-1)

GR = GR.fit(X_train, y_train)

```

The number of nodes and the maximum depth of the tree.

```

: GR.best_estimator_.tree_.node_count, GR.best_estimator_.tree_.max_depth
: (99, 7)

```

remarks
param_grid (dict)
scoring='accuracy' (gridsearch need to maximize a value,
here need to max 'accuracy')
n_jobs= -1 (paralyze across all of the power, all the
functionality that's available on our computer)

DecisionTree Regressor

```

from sklearn.tree import DecisionTreeRegressor

dr = DecisionTreeRegressor().fit(X_train, y_train)

param_grid = {'max_depth':range(1, dr.tree_.max_depth+1, 2),
              'max_features': range(1, len(dr.feature_importances_)+1)}

GR_sugar = GridSearchCV(DecisionTreeRegressor(random_state=42),
                       param_grid=param_grid,
                       scoring='neg_mean_squared_error',
                       n_jobs=-1)

GR_sugar = GR_sugar.fit(X_train, y_train)

```

The number of nodes and the maximum depth of the tree. This tree has lots of nodes, which is not so surprising given the continuous data.

```

GR_sugar.best_estimator_.tree_.node_count, GR_sugar.best_estimator_.tree
(7475, 23)

```

remarks
scoring='neg_mean_squared_error'
(gridsearch need to maximize a value, here need to

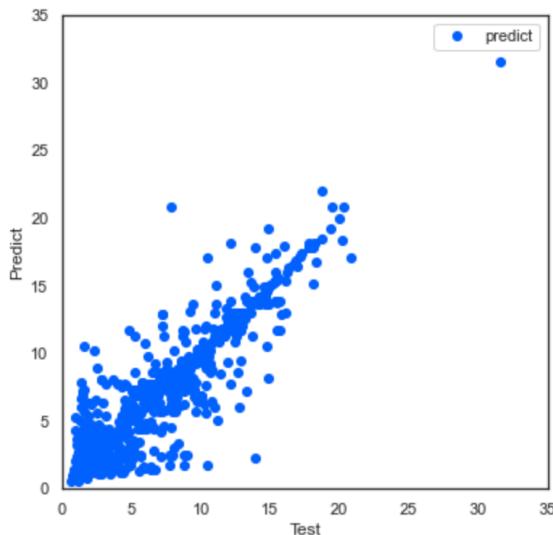
minimize the MSE, so equals to maximize neg_MSE)

Plot actual vs. predicted

```
sns.set_context('notebook')
sns.set_style('white')
fig = plt.figure(figsize=(6,6))
ax = plt.axes()

ph_test_predict = pd.DataFrame({'test':y_test.values,
                                 'predict': y_test_pred_gr_sugar}).set_index('test').sort_index()

ph_test_predict.plot(marker='o', ls='', ax=ax)
ax.set(xlabel='Test', ylabel='Predict', xlim=(0,35), ylim=(0,35));
### END SOLUTION
```



If actual = predicted, 应该是一条对角线。

Plot the TREE (not prune vs. prune)

```
from io import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
!pip install pydotplus
```

StringIO (we can dump out the file as a string)

Image (show image within Jupiter notebook)

sklearn.tree — export_graphviz (allow to take a graph which is our learned model and export into a file)

pydotplus (will take that file and allow us to render an actual image)

Not prune the tree: (original dt model)

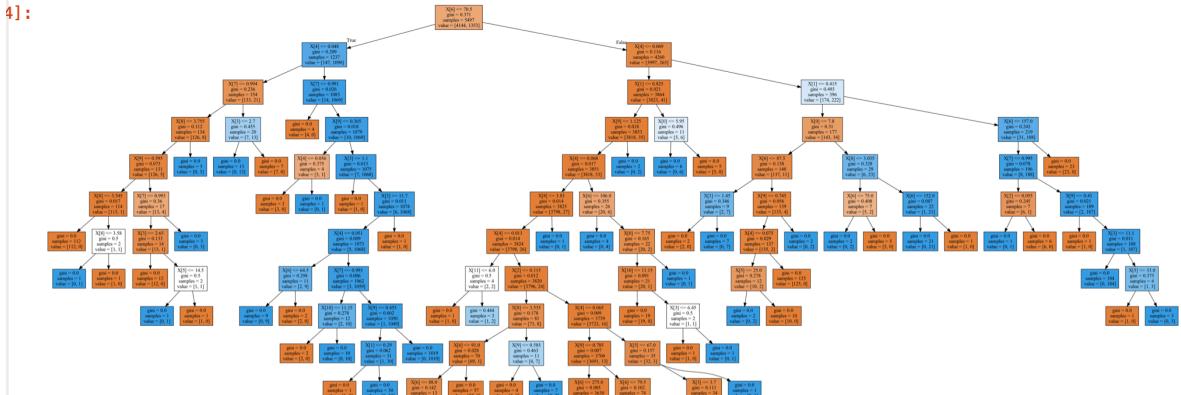
```

4]: ### BEGIN SOLUTION
# Create an output destination for the file
dot_data = StringIO()

export_graphviz(dt, out_file=dot_data, filled=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())

# View the tree image
filename = 'wine_tree_no_prune.png'
graph.write_png(filename)
Image(filename=filename)

```



Prune the tree (by grid search model)

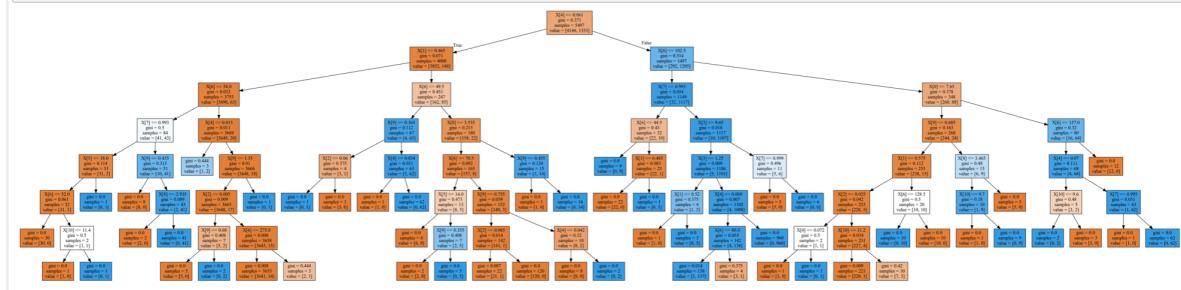
```

# Create an output destination for the file
dot_data = StringIO()

export_graphviz(GR.best_estimator_, out_file=dot_data, filled=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())

# View the tree image
filename = 'wine_tree_prune.png'
graph.write_png(filename)
Image(filename=filename)
### END SOLUTION

```



Wk3_Ensemble Based Methods and Bagging (2022.01.06)

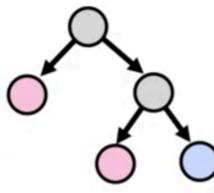
Tree ensembles have been found to generalize well when scoring new data. Some useful and popular tree ensembles are **bagging**, **boosting**, and **random forests**.

Bagging, which combines decision trees by using bootstrap aggregated samples.

An **advantage** specific to bagging is that this method can be multithreaded or computed in parallel. Most of these ensembles are assessed using **out-of-bag error**.

Bagging Error Calculations

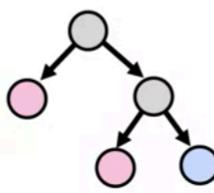
Date	Title	Budget	Domestic/Total/Gross	Director	Rating	RunTime
8/2013-08-01	The Hunger Games: Catching Fire	120000000	488000000	François Leterrier	PG-13	140
1/2013-01-03	Iron Man 3	200000000	1000000000	Jon Favreau	PG-13	140
2/2013-01-22	Prisoners	100000000	403790000	Chris Columbus	PG	125
3/2013-03-14	Despicable Me 2	700000000	2800000000	Pierre Coffin/Chris Renaud	PG	98
4/2013-04-04	Man of Steel	220000000	2191000000	Zack Snyder	PG-13	140
5/2013-05-10	Gravity	100000000	1740000000	Alfonso Cuarón	PG-13	91
6/2013-06-21	Monsters University	N/A	288000000	Dan Scanlon	G	107
7/2013-07-12	Now You See Me	100000000	1000000000	Neill Blomkamp	PG-13	130
8/2013-08-08	Fruit & Purves S	100000000	288000000	Peter Jackson	PG-13	130
9/2013-09-12	On The Great and Powerful	210000000	236011803	Sam Raimi	PG	127
10/2013-10-18	Star Trek Into Darkness	180000000	228719801	J.J. Abrams	PG-13	123
11/2013-11-08	Thor: The Dark World	170000000	2020000000	Alan Taylor	PG-13	140
12/2013-12-01	World War Z	180000000	1000000000	Marc Forster	PG-13	118
1/2014-01-06	Insurgent	100000000	1000000000	Doug Liman	PG-13	118
2/2014-02-07	The Heat	400000000	2100000000	Paul Feig	PG	117
3/2014-03-07	We're the Millers	270000000	1000000000	Reagan Marshall-Thomar	R	110
4/2014-04-11	American Hustle	400000000	100117807	David O. Russell	R	138
5/2014-05-15	The Great Gatsby	100000000	14484518	Baz Luhrmann	PG-13	140



Same as decision trees:

- Easy to interpret and implement
- Heterogeneous input data allowed, no preprocessing required

Date	Title	Budget	Domestic/Total/Gross	Director	Rating	RunTime
8/2013-08-01	The Hunger Games: Catching Fire	120000000	488000000	François Leterrier	PG-13	140
1/2013-01-03	Iron Man 3	200000000	1000000000	Jon Favreau	PG-13	140
2/2013-01-22	Prisoners	100000000	403790000	Chris Columbus	PG	125
3/2013-03-14	Despicable Me 2	700000000	2800000000	Pierre Coffin/Chris Renaud	PG	98
4/2013-04-04	Man of Steel	220000000	2191000000	Zack Snyder	PG-13	140
5/2013-05-10	Gravity	100000000	1740000000	Alfonso Cuarón	PG-13	91
6/2013-06-21	Monsters University	N/A	288000000	Dan Scanlon	G	107
7/2013-07-12	Now You See Me	100000000	1000000000	Neill Blomkamp	PG-13	130
8/2013-08-08	Fruit & Purves S	100000000	288000000	Peter Jackson	PG-13	130
9/2013-09-12	On The Great and Powerful	210000000	236011803	Sam Raimi	PG	127
10/2013-10-18	Star Trek Into Darkness	180000000	228719801	J.J. Abrams	PG-13	123
11/2013-11-08	Thor: The Dark World	170000000	2020000000	Alan Taylor	PG-13	140
12/2013-12-01	World War Z	180000000	1000000000	Marc Forster	PG-13	118
1/2014-01-06	Insurgent	100000000	1000000000	Doug Liman	PG-13	118
2/2014-02-07	The Heat	400000000	2100000000	Paul Feig	PG	117
3/2014-03-07	We're the Millers	270000000	1000000000	Reagan Marshall-Thomar	R	110
4/2014-04-11	American Hustle	400000000	100117807	David O. Russell	R	138
5/2014-05-15	The Great Gatsby	100000000	14484518	Baz Luhrmann	PG-13	140



Specific to bagging:

- Less variability than decision trees
- Can grow trees in parallel

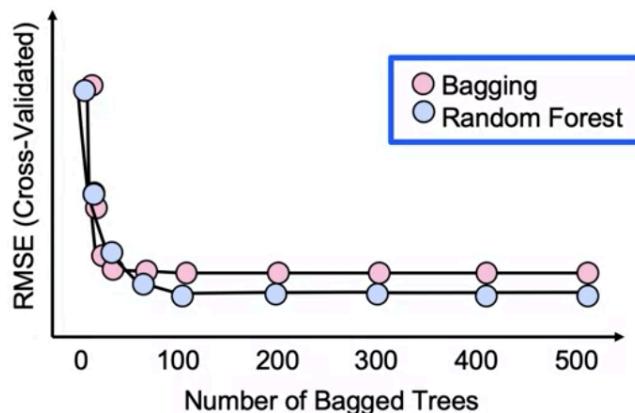
Random Forest

Random forest is a tree ensemble that has a similar approach to bagging. Their main characteristic is that they add randomness by only using a subset of features to train each split of the trees it trains.

Extra Random Trees is an implementation that adds randomness by creating splits at random, instead of using a greedy search to find split variables and split points.

Introducing More Randomness

- Solution: further de-correlate trees.
- Use random subset of features for each tree.
 - Classification: \sqrt{m}
 - Regression: $m/3$
- Called "Random Forest".



Code

```
# Import the class containing the classification method
from sklearn.ensemble import RandomForestClassifier

# Create an instance of the class
RC = RandomForestClassifier(n_estimators=50)

# Fit the instance on the data and then predict the expected value
RC = RC.fit(X_train, y_train)
y_predict = RC.predict(X_test)
```

Tune parameters with cross-validation. Use `RandomForestRegressor` for regression.

Code

```
# Import the class containing the classification method
from sklearn.ensemble import ExtraTreesClassifier

# Create an instance of the class
EC = ExtraTreesClassifier(n_estimators=50)

# Fit the instance on the data and then predict the expected value
EC = EC.fit(X_train, y_train)
y_predict = EC.predict(X_test)
```

Tune parameters with cross-validation. Use `ExtraTreesRegressor` for regression.

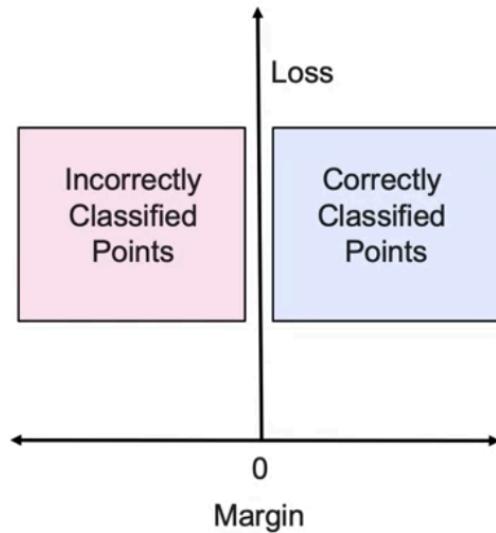
Boosting

Boosting methods are **additive** in the sense that they sequentially retrain decision trees using the observations with the highest residuals on the previous tree.

To do so, observations with a high residual are assigned a higher **weight**.

Boosting Specifics

- Boosting utilizes different loss functions.
- At each stage, the margin is determined for each point.
- Margin is positive for correctly classified points and negative for misclassifications.
- Value of loss function is calculated from margin.

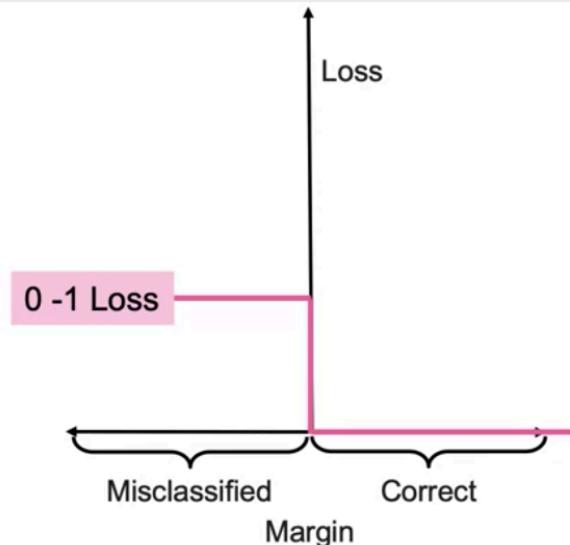


Gradient Boosting

The main **loss functions** for boosting algorithms are:

- **0-1 loss function**, which ignores observations that were correctly classified. The shape of this loss function makes it difficult to optimize.

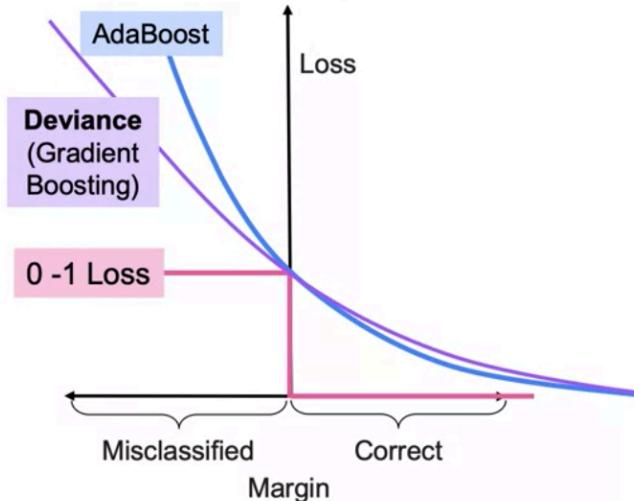
- The 0 – 1 Loss multiplies misclassified points by 1.
- Correctly classified points are ignored.
- Theoretical "ideal" loss function.
- Difficult to optimize
 - non-smooth and non-convex.



- **Adaptive boosting loss function**, which has an exponential nature. The shape of this function is more sensitive to outliers.

Gradient Boosting Loss Function

- Generalized boosting method that can use different loss functions.
 - Common implementation uses binomial log likelihood loss function (**deviance**):
- $$\log(1 + e^{-\text{margin}})$$
- More robust to outliers than AdaBoost.



- **Gradient boosting loss function**. The most common gradient boosting implementation uses a binomial log-likelihood loss function called **deviance**. It tends to be more robust to outliers than AdaBoost.

Bagging vs Boosting

Bagging

- Bootstrapped samples
- Base trees created independently
- Only data points considered
- No weighting used
- Excess trees will not overfit

Boosting

- Fit entire data set
- Base trees created successively
- Use residuals from previous models
- Up-weight misclassified points
- Beware of overfitting

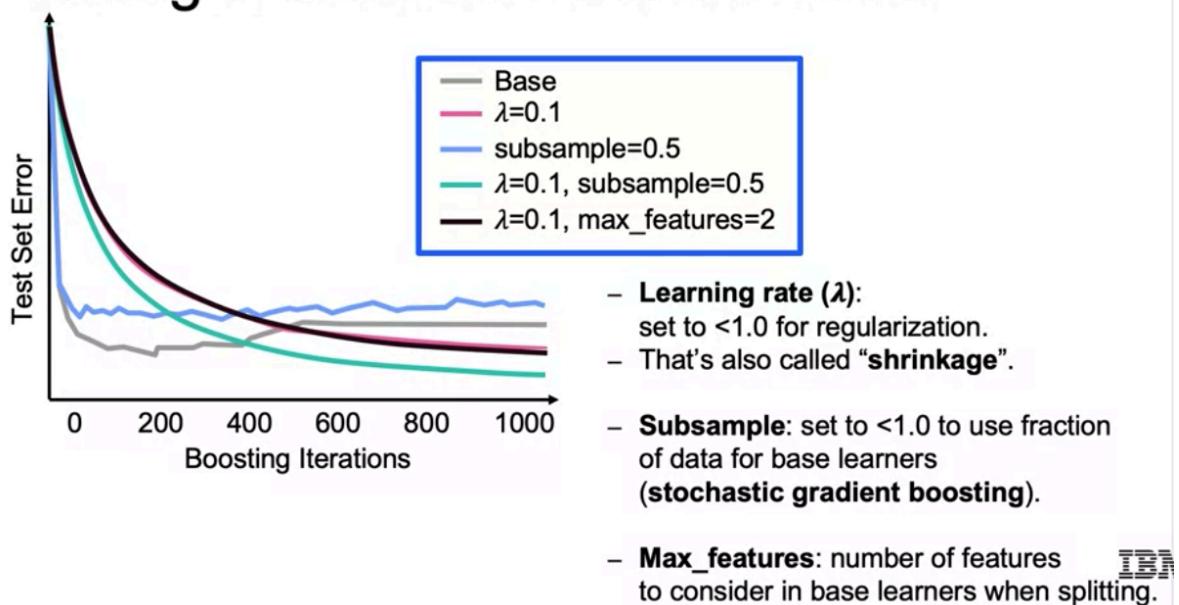
The additive nature of gradient boosting makes it **prone to overfitting**.

This can be addressed using **cross validation** or **fine tuning** the number of boosting iterations.

Other hyperparameters to fine tune are:

- learning rate (shrinkage). <1
- subsample. 0.5
- number of features. max_features

Tuning a Gradient Boosted Model



Stacking

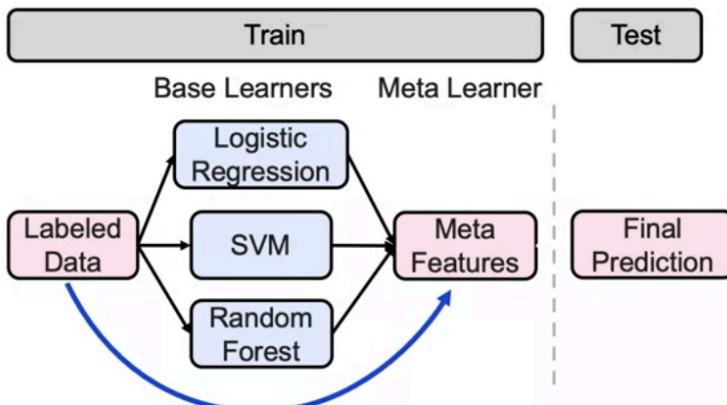
Stacking is an ensemble method that combines any type of model by combining the predicted probabilities of classes.

In that sense, it is a generalized case of bagging.

The two most common ways to combine the predicted probabilities in stacking

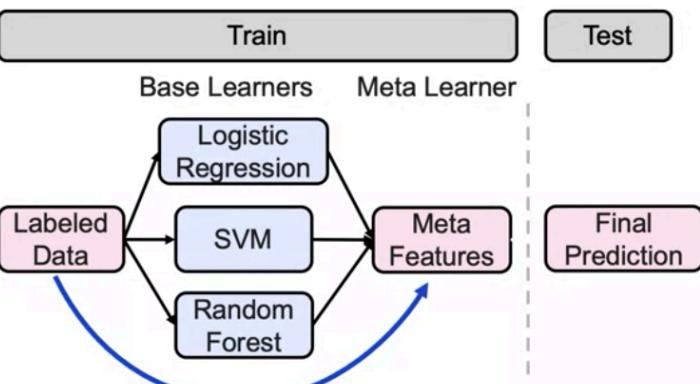
are: using a **majority vote** or using **weights** for each predicted probability.

Stacking: Combining Classifiers



- Models of any kind can be combined to create a stacked model.
- Like bagging but not limited to decision trees.
- Output of base learners creates features, can recombine with data.

Stacking: Combining Classifiers



- Output of base learners can be combined via majority vote or weighted.
- Additional hold-out data needed if meta learner parameters are used.
- Be aware of increasing model complexity.
- The final prediction can be done by voting or with another model

Code

```
# Import the class containing the classification method
from sklearn.ensemble import VotingClassifier

# Create an instance of the class
VC = VotingClassifier(estimator_list)

# Fit the instance on the data and then predict the expected value
VC = VC.fit(X_train, y_train)
y_predict = VC.predict(X_test)

Use VotingRegressor for regression.

The StackingClassifier (or StackingRegressor) works similarly:
SC = StackingClassifier(estimator_list, final_estimator=LogisticRegression())
```

In this section, we discussed:

- The Boosting approach to combining models
- Types of Boosting models: Gradient Boosting, AdaBoost
- Boosting loss functions
- Combining heterogeneous classifiers

Further reading:

- XGBoost is another popular boosting algorithm (not in Scikit-Learn).

Exam (Check every question!!)

← 首页 End of Module
预计用时>10 min

恭喜！您通过了！
获得的成绩 100% 通过条件 66% 或更高

End of Module
最新提交作业的评分 100%

1. The term Bagging stands for bootstrap aggregating.
 True
 False
 正确
Correct! You can find more information in the lesson: Ensemble Based Methods and Bagging.

2. This is the best way to choose the number of trees to build on a Bagging ensemble.
 Choose a large number of trees, typically above 100
 Prioritize training error metrics over out-of-bag sample
 Tune number of trees as a hyperparameter that needs to be optimized
 Choose a number of trees past the point of diminishing returns
 正确
Correct! You can find more information in the lesson: Ensemble Based Methods and Bagging.

3. Which type of Ensemble modeling approach is NOT a special case of model averaging?
 Random Forest methods
 Boosting methods
 The Pasting method of Bootstrap aggregation
 The Bagging method of Bootstrap aggregation
 正确
Correct! You can find more information in the lesson: Overview of Boosting.

4. What is an ensemble model that needs you to look out of bag error?
 Logistic Regression
 Out of Bag Regression
 Random Forest
 Stacking
 正确
Correct! You can find more information in the lesson: Random Forest.

5. What is the main condition to use stacking as ensemble method?
 Models need to be parametric
 Models need to be nonparametric
 Models need to output residual values for each class
 Models need to output predicted probabilities
 正确
Correct! You can find more information in the lesson: Stacking.

6. This tree ensemble method only uses a subset of the features for each tree.
 Random Forest
 AdaBoost
 Bagging
 Stacking
 正确
Correct! This tree ensemble only uses a subset of the features for each tree. For more information, please review the Random Forest lesson.

7. Order these tree ensembles in order of most randomness to least randomness.
 Random Forest, Random Trees, Bagging
 Bagging, Random Forest, Random Trees
 Random Trees, Random Forest, Bagging
 Random Forest, Bagging, Random Trees
 正确
Correct! Random Trees add one more degree of randomness than Random Forests and less than Bagging. You can find more information in the Random Forest lesson.

8. This is an ensemble model that does not use bootstrapped samples to fit the base trees, takes residuals into account, and fits the base trees iteratively.
 Random Trees
 Boosting
 Bagging
 Random Forest
 正确
Correct! These are all characteristics of boosting algorithms. You can find more information in the Boosting lesson.

Wk4_Unbalanced Classes (2022.01.07)

Modeling Unbalanced Classes

Classification algorithms are built to optimize accuracy, which makes it challenging to create a model when there is not a balance across the number of observations of different classes.

Common methods to approach balancing the classes are:

- **Downsampling** or removing observations from the most common class

Downsampling adds tremendous importance to the minor class, typically shooting up recall and bringing down precision.

Values like 0.8 recall and 0.15 precision isn't uncommon.

- **Upsampling** or duplicating observations from the rarest class or classes

Upsampling mitigates some of the excessive weight on the minor class. Recall is still typically higher than precision, but the gap is lesser.

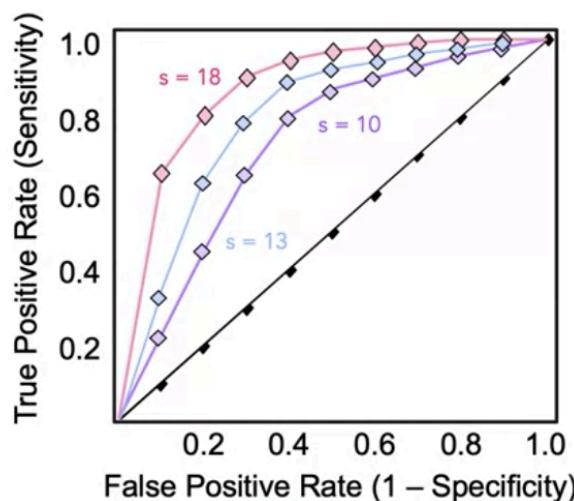
Values like 0.7 recall and 0.4 precision isn't uncommon. And are often considered good results for an unbalanced dataset.

- A **mix** of downsampling and upsampling

Steps for unbalanced datasets:

- Do a stratified test-train split
- Up or down sample the full dataset
- Build models

Cross-validation works for any global model-making choice, including sampling.



Modeling Approaches for Unbalanced Classes

Specific algorithms to upsample and downsample are:

- **Stratified sampling**

Stratified Sampling

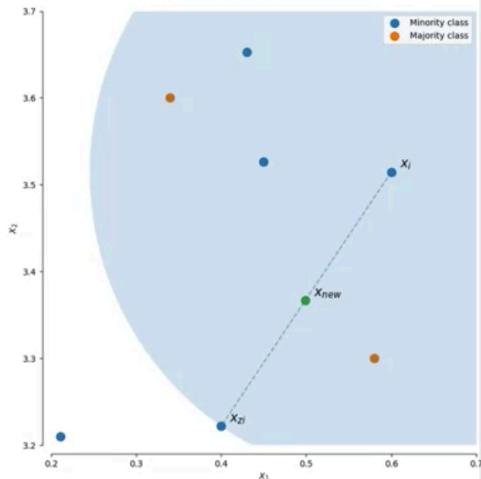
- Train-test split, “stratify” option
 - ShuffleSplit -> StratifiedShuffleSplit
 - KFold -> StratifiedKFold -> RepeatedStratifiedKFold
- Random oversampling

Random Oversampling

- Simplest oversampling approach
 - Resample with replacement from minority class
 - No concerns about geometry of feature space
 - Good for categorical data
- Synthetic oversampling,
 - the main two approaches being :
 - ◆ Synthetic Minority Oversampling Technique (**SMOTE**) and;
 - ◆ Adaptive Synthetic sampling (**ADASYN**)

Synthetic Oversampling

- Start with a point in the minority class
- Choose one of K nearest neighbors
- Add a new point between them
- Two main approaches:
 - SMOTE
 - ADASYN



http://contrib.scikit-learn.org/imbalanced-learn/stable/over_sampling.html#a-practical-guide

SMOTE: Synthetic Minority Oversampling Technique

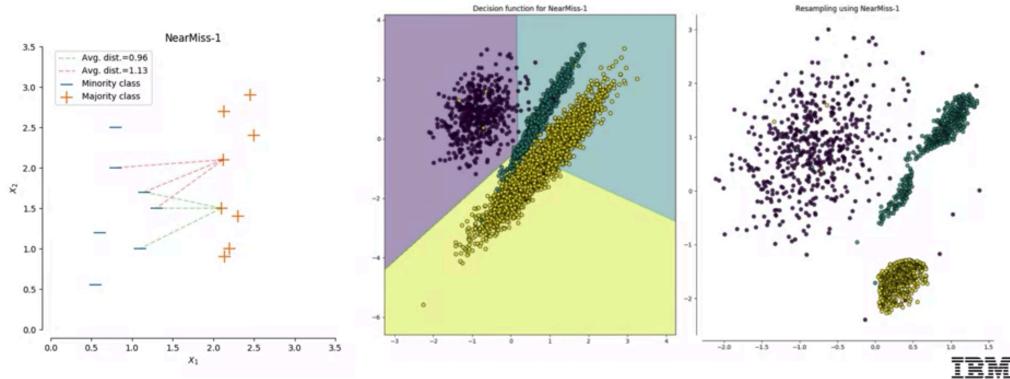
- **Regular:** Connect minority class points to any neighbor (even other classes)
- **Borderline:** Classify points as outlier, safe, or in-danger
 - 1: Connect minority in-danger points only to minority points
 - 2: Connect minority in-danger points to whatever is nearby
- **SVM:** Use minority support vectors to generate new points

ADASYN: ADAptive SYNthetic sampling

- For each minority point:
 - Look at classes in neighborhood
 - Generate new samples proportional to competing classes
- Motivated by KNN, but helps other classifiers as well
- Cluster Centroids implementations like:
 - ◆ NearMiss 1,2,3
 - ◆ Tomek Links, and;
 - ◆ Nearest Neighbors

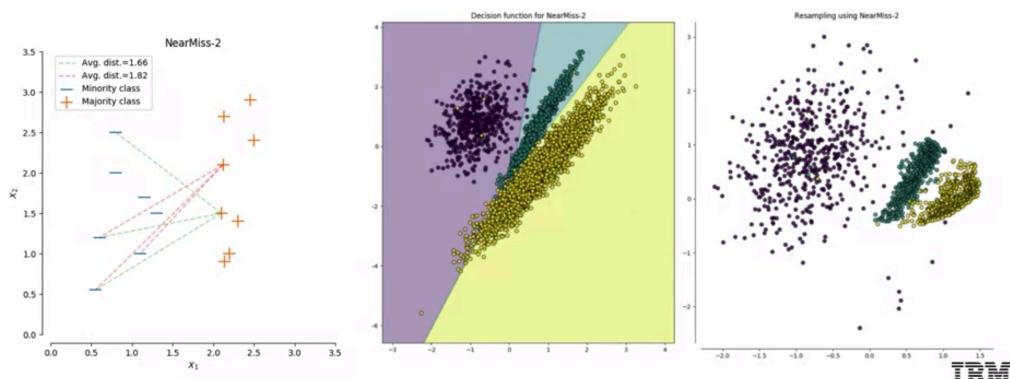
imbalanced-learn documentation:
<https://imbalanced-learn.org/stable/index.html>

NearMiss-1: Which Points to Keep? Those Closest to Nearby Minority Points



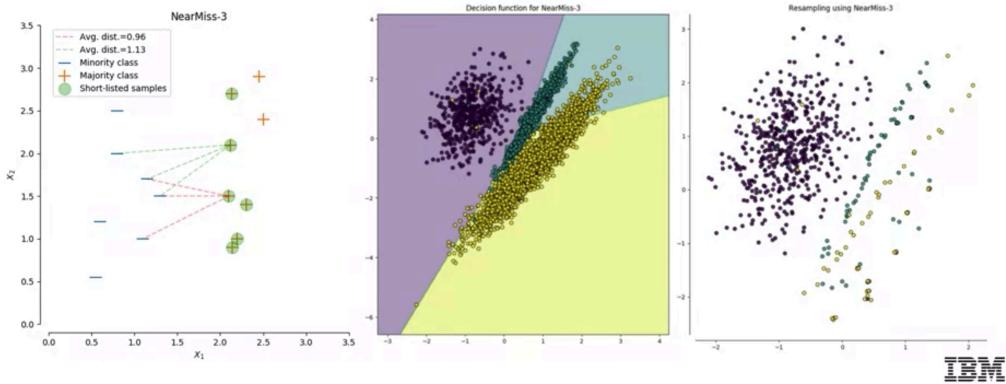
IBM

NearMiss-2: Which Points to Keep? Those Closest to Distant Minority Points



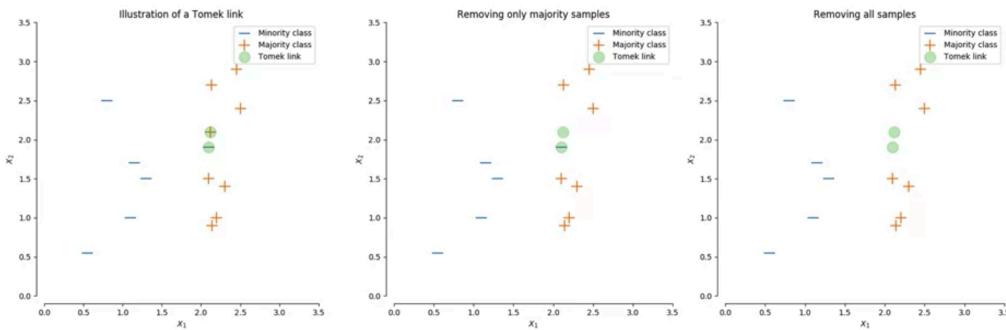
IBM

NearMiss-3: Those Closest to Majority Neighbors of Minority Points



IBM

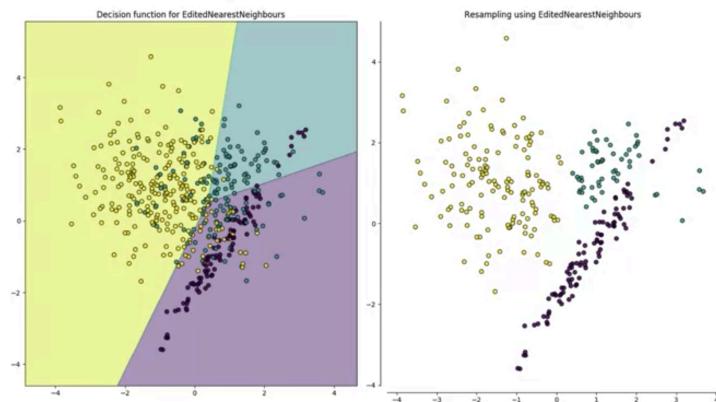
Tomek Links: Mixed Mutual Nearest Neighbors



IBM

Edited Nearest Neighbors

- Remove points that don't agree with neighbors



IBM

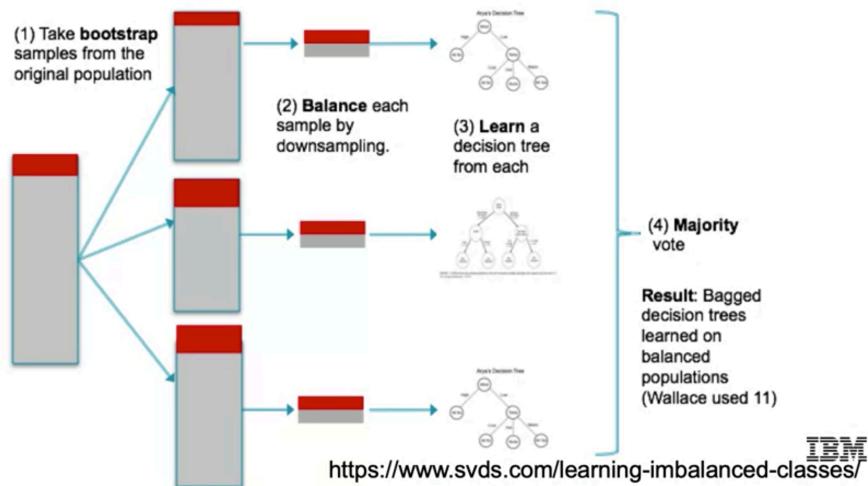
Combination Over/Under

- SMOTE + Tomek's link
- SMOTE + Edited Nearest Neighbors



IBM

Blagging (Balanced Bagging)



Unbalanced Classes: Summary

- All of this happens after the test set has been split.
- Use sensible metrics
 - AUC
 - F1
 - Cohen's Kappa
- Not accuracy - too easy to fool in this case

Now that first step is going to ensure that you first do your train test split before doing any of this over or undersampling. Recall that if we do this oversampling first, we can end up with values being both represented in the train and test set, which means we can very easily overfit, have that data leakage and even if we're synthetically creating those new samples, those values will be very close to those train set samples and we can still have that overfitting. So always do your train test split first.



IBM

Learning Recap

In this section, we discussed:

- Additional approaches to dealing with unbalanced outcomes
- Random and Synthetic Over Sampling
- Techniques for Under Sampling
- Using Balanced Bagging (Bagging) to address unbalanced class data

Note: I want you to note here that for both of them, they each have their advantages and practices for each one of them, for the specific techniques for each of them. But it'll often be difficult to tell for oversampling or undersampling which technique to use without cross-validation since we can almost never visualize the shape of the data. But when deciding between oversampling and undersampling, remember oversampling in general, remember what we discussed earlier. Undersampling will probably lead to a bit of a higher recall for that minority class at the cost of precision whereas oversampling will keep all the values from our majority class and thus will have a bit of a lower recall than undersampling, but better precision on those predictions.



◀ 返回 Modeling Unbalanced Classes
练习测验 • 4 min

⚠ 准备好后再次尝试
获得的成绩 50% 通过条件 66% 或更高 再试

Modeling Unbalanced Classes
总分 2

1. These are all methods of dealing with unbalanced classes EXCEPT:
1/1 分

Downsampling.
 Mix of in-sample and out-of-sample.
 Mix of downsampling and upsampling.
 Upsampling.

(正确) Correct! You can find more information in the lesson *Upsampling and Downsampling*.

2. (True/False) A best practice to build a model using unbalanced classes is to split the data first, then apply an upsample or undersample technique.
0/1 分

True
 False

(错误) Incorrect. A best practice is to do a stratified train/test split before, then use an upsample or downsample technique, and last build a predictive model. You can find more information in the lesson *Upsampling and Downsampling*.