Machine Learning and Artificial Intelligence

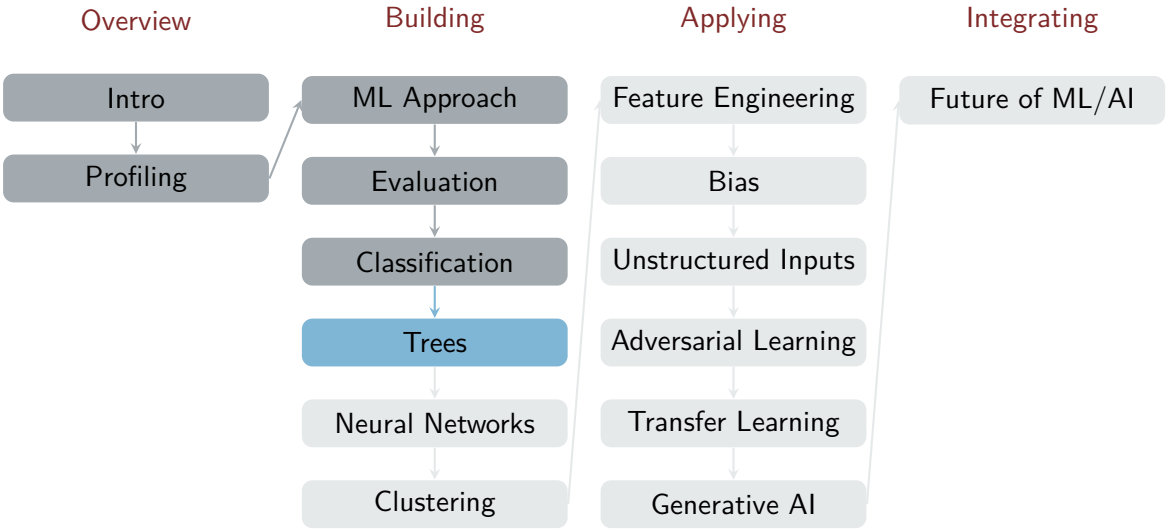
Session 8: Trees

Do Yoon Kim

Boston College

14 February 2024

Course Roadmap



Today

Learning Objectives

- Build multiple tree models.
- Compare / contrast different tree models.
- Practice models.

Agenda

1. Decision Trees
2. Bagged Trees
3. Random Forests
4. Boosted Trees
5. Tuning

Administrative notes...

- Managerial project: Presentation schedule coming this weekend.
- Code project scope: Starting to go through them now.
- Any questions?

THE WALL STREET JOURNAL.

Sam Altman Seeks Trillions of Dollars to Reshape Business of Chips and AI

OpenAI chief pursues investors including the U.A.E. for a project possibly requiring up to $7 trillion

The amounts Altman has discussed would also be outlandishly large by the standards of corporate fundraising—larger than the national debt of some major global economies and bigger than giant sovereign-wealth funds.

Such a sum of investment would dwarf the current size of the global semiconductor industry. Global sales of chips were $527 billion last year and are expected to rise to $1 trillion annually by 2030. Global sales of semiconductor manufacturing equipment—the costly machinery needed to run chip factories—last year were $100 billion, according to an estimate by the industry group SEMI.

https://www.wsj.com/tech/ai/sam-altman-seeks-trillions-of-dollars-to-reshape-business-of-chips-and-ai-89ab3db0

The Guardian

AI helps scholars read scroll buried when Vesuvius erupted in AD79

Scholars of antiquity believe they are on the brink of a new era of understanding after researchers armed with artificial intelligence read the hidden text of a charred scroll that was buried when Mount Vesuvius erupted nearly 2,000 years ago.

The breakthrough in reading the ancient material came from the $1m Vesuvius Challenge, a contest launched in 2023 by Brent Seales, a computer scientist at the University of Kentucky, and Silicon Valley backers. The competition offered prizes for extracting text from high-resolution CT scans of a scroll taken at Diamond, the UK's national synchrotron facility in Oxfordshire.

https://www.theguardian.com/science/2024/feb/05/ai-helps-scholars-read-scroll-buried-when-vesuvius-erupted-in-ad79

THE VERGE

These $349 smart glasses have 'AI superpowers' and a comical charging nose

There's a new pair of smart spectacles in town — and this time, it's the $349 Frame glasses said to give you multimodal "AI superpowers." The open-source eyewear comes from a startup called Brilliant Labs, which touts Frame as a way to get AI translations, web search, and visual analysis right in front of your eyes.

As shown in a video posted by Brilliant Labs, you can use your voice to ask the glasses to do things like identify a landmark you're looking at, search the web for a particular pair of sneakers you're seeing, or even look up nutrition information for the food you're about to eat. The information appears as an overlay that shows up directly on the lens.

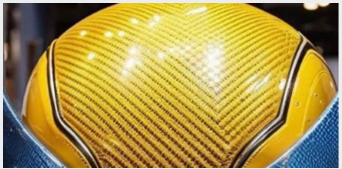
Frame pairs with Brilliant Labs' app, called Noa. The app contains an AI assistant that uses OpenAI for visual analysis, Whisper for translation, and Perplexity for web search. In an interview with Venture Beat, Brilliant Labs says its Noa AI "learns and adapts to both the user and the tasks it receives."

https://www.theverge.com/2024/2/9/24067485/frame-ai-smart-glasses-brilliant-labs

Forbes — Deadpool 3's 'Leaked' Wolverine Mask Feels Like A Watershed Moment For AI Lies

This image was widely shared on social media over the last couple days, and at first, not really scrutinized at all... It looks pretty good, I thought, before making my way into the replies.

We are getting to the point where you genuinely have to examine things under a microscope to try and find clues, and even knowing you need to do that, I still couldn't see the telltale signs of AI until an army of people banded together to hunt them down.



The off-center pattern on the forehead:

Things like the pointy nose shadow not matching the actual nose:

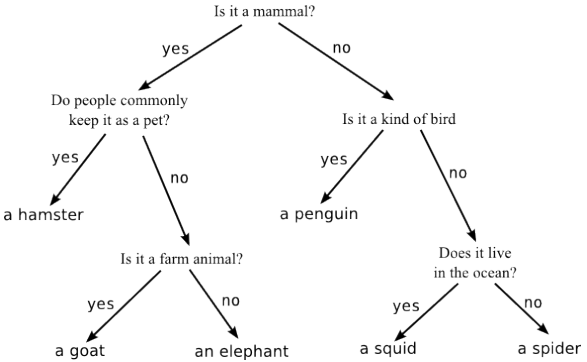
Some of the signs appearing to be nonsense

Customers with credit cards can either make their payment or miss their payment. Managerially, it would be really help to know which customers will miss their payment. The "creditCardDefault.csv" file contains cases from customers in Taiwan. We'll consider different models to classify customers and how to compare them.

- Use script from last time. We'll add to it.
- Import data. Split into test/train.
- Features:
  - Numeric ('Limit_Bal', 'Age', 'Bill_Amt1', 'Pay_Amt1') with StandardScaler.
  - Categorical ('Card', 'Marriage', 'Pay_0') with OneHotEncoder.

Get everything loaded and where we left off.

Decision Trees



https://lelandkrych.wordpress.com/2016/05/10/machine-learning-for-data-analysis-decision-trees/

- A flowchart-like structure
  - Each internal node represents a test on an attribute.
  - Each branch represents the outcome of the test.
  - Each leaf node represents a class label .
  - The paths from root to leaf represent classification rules.
- Pros and cons?
  - ✔ Simple
  - ✔ White-box
  - ✗ Sensitive to data ('high variance')
  - ✗ May require many levels
  - ✗ Generally low accuracy

? Make a pipeline that uses `DecisionTreeClassifier` from the `sklearn.tree` package.

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> pipeDt = Pipeline([
        ('preprocessor', preprocessor ),
        ('model',        DecisionTreeClassifier() )
    ], verbose=True )
```

You should be getting pretty comfortable with this part of the process. What is next?

Fit the decision tree model on the training data.

```
>>> pipeDt.fit( X_train, y_train )
```

Use the fitted model to predict values in training and in test.

```
>>> predTrainDt = pipeDt.predict( X_train )
>>> predTestDt = pipeDt.predict( X_test )
```

Assess the fit in training.

```
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainDt )
0.997556867026161
```

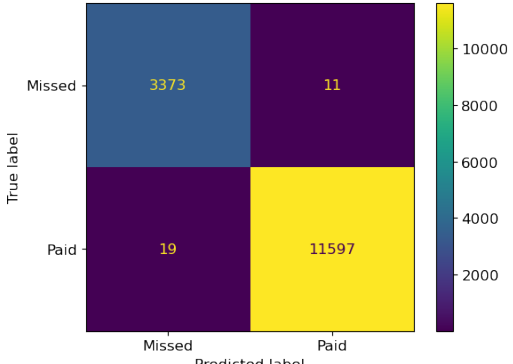
Look at the classification report.

```
>>> print( sklearn.metrics.classification_report( y_train, predTrainDt ) )
              precision    recall  f1-score   support

      Missed       0.99      1.00      1.00      3384
        Paid       1.00      1.00      1.00     11616

    accuracy                           1.00     15000
   macro avg       1.00      1.00      1.00     15000
weighted avg       1.00      1.00      1.00     15000
```

That looks like a pretty sweet model!

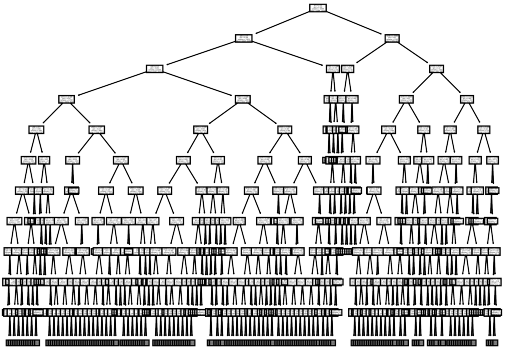
Check the confusion matrix.

```
>>> cmDtTrain = sklearn.metrics.confusion_matrix( y_train, predTrainDt )
>>> plotCmDtTrain = sklearn.metrics.ConfusionMatrixDisplay( cmDtTrain, display_labels=pipeDt.classes_ ).
    plot()
```



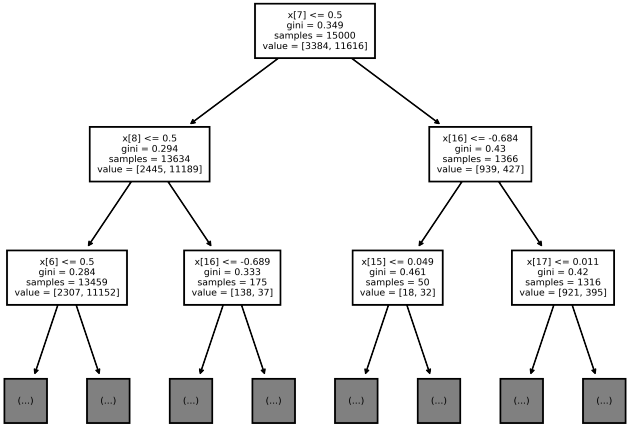
? A nice thing about decision trees is that humans can understand how they work. We can use the `plot_tree` function in `sklearn.tree` to see it.

```
>>> from sklearn.tree import plot_tree
>>> plotTree = sklearn.tree.plot_tree( pipeDt['model'], max_depth=10 )
```



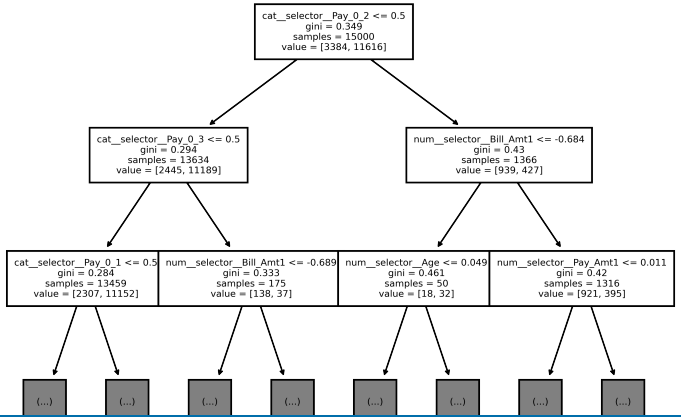
If we restrict the depth to 2 levels, we can read it better.

```
>>> plotTree = sklearn.tree.plot_tree( pipeDt['model'], max_depth=2 )
```



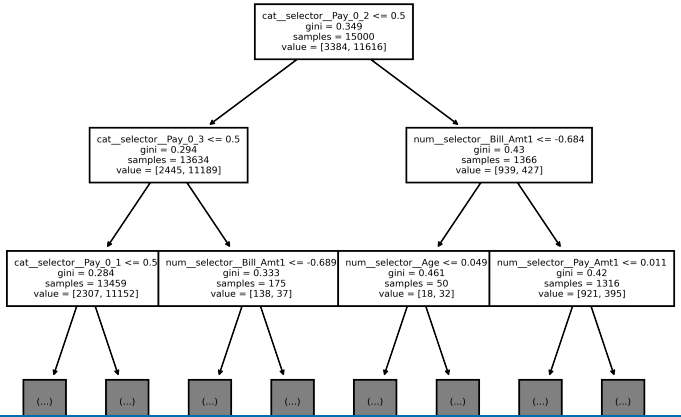
? Features names are hard to figure out.

```
>>> tempFeatures = pipeDt[:-1].get_feature_names_out().tolist() # weird syntax gets names from last step
>>> plotTree = sklearn.tree.plot_tree( pipeDt['model'], max_depth=2, feature_names = tempFeatures )
```



Features names are hard to figure out.

```
>>> tempFeatures = pipeDt[:-1].get_feature_names_out() # this weird syntax gets names from the last step
>>> plotTree = sklearn.tree.plot_tree( pipeDt['model'], max_depth=2, feature_names = tempFeatures )
```



❓ Check the fit in the testing data.

```
>>> sklearn.metrics.balanced_accuracy_score( y_test, predTestDt )
0.6018565458892587
```

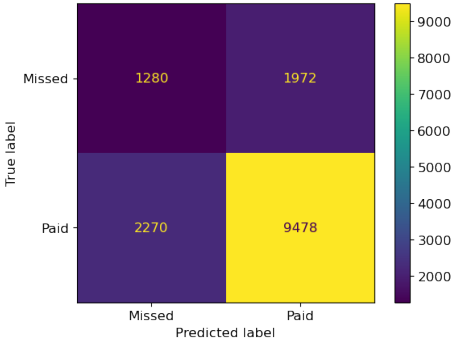
❓ Classification report for test data?

```
>>> print( sklearn.metrics.classification_report( y_test, predTestDt ) )
              precision    recall  f1-score   support

      Missed       0.36      0.39      0.38      3252
        Paid       0.83      0.81      0.82     11748

    accuracy                           0.72     15000
   macro avg       0.59      0.60      0.60     15000
weighted avg       0.73      0.72      0.72     15000
```

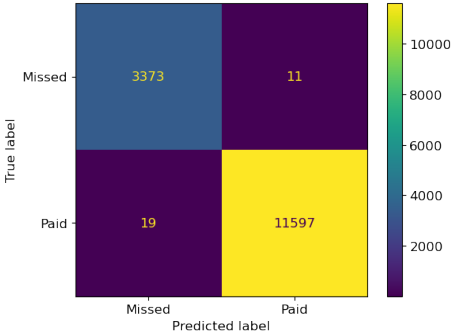
? Confusion matrix for test data?

```
>>> cmDtTest = sklearn.metrics.confusion_matrix( y_test, predTestDt )
>>> plotCmDtTest = sklearn.metrics.ConfusionMatrixDisplay( cmDtTest, display_labels=pipeDt.classes_  ).
    plot()
```

Test data



Training data

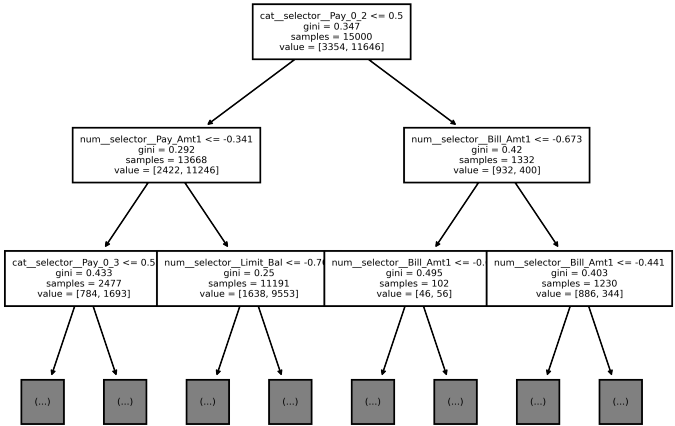


But really sensitive to sample and order

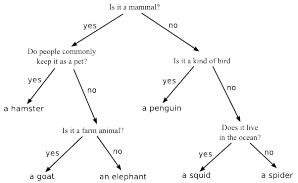
(Illustration only; you don't need to do this.)

```
df2 = df.sample(frac = 1)
X_train2, X_test2, y_train2, y_test2 = train_test_split( df2, df2.Payment, test_size=0.5 )
pipeDt2 = pipeDt
pipeDt2.fit( X_train2, y_train2 )

predTrainDt2 = pipeDt2.predict( X_train2 )
sklearn.metrics.balanced_accuracy_score( y_train2, predTrainDt2 )
print( sklearn.metrics.classification_report( y_train2, predTrainDt2 ) )

cmDtTrain2 = sklearn.metrics.confusion_matrix( y_train2, predTrainDt2 )
tempFeatures2 = pipeDt2[:-1].get_feature_names_out()
plotTree2 = sklearn.tree.plot_tree( pipeDt2['model'], max_depth=2, feature_names = tempFeatures2,
     fontsize=5 )
```

But really sensitive to sample and order



Decision Trees — what do we observe?



- ✔ Simple
- ✔ White-box
- ✗ Requires many levels
- ✗ Easy to overfit.
- ✗ Sensitive to data ('high variance').
  - We would get a different decision tree if we sampled the data.
  - We would get a different decision tree if we sorted the data.

Hmm...we could use this sensitivity to our advantage. What if we make a whole bunch of decision trees, then somehow average them???

https://lelandkrych.wordpress.com/2016/05/10/machine-learning-for-data-analysis-decision-trees/

Bagged Trees

Like decision tree but...

- Bootstraping randomly picks observations.
- Some observations picked, others not.
- Some observations picked many times.
- Build a tree with each selection.
- Each tree may overfit but we average many trees.

✗ But harder to interpret.

A `BaggingClassifier` will sample and average for us.

```
>>> from sklearn.ensemble import BaggingClassifier
>>> pipeBagDt = Pipeline([
        ('preprocessor', preprocessor ),
        ('model',        BaggingClassifier( DecisionTreeClassifier() ) )
    ], verbose=True )
```

Notice that this syntax differs. You tell the `BaggingClassifier` what kind of model to repeatedly sample. But it still works in the pipeline approach.


```
>>> pipeBagDt.fit( X_train, y_train )
```


```
>>> predTrainBagDt = pipeBagDt.predict( X_train )
>>> predTestBagDt = pipeBagDt.predict( X_test )
```

? Compare the accuracy to the decision tree accuracy in training.

```
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainDt )
0.997556867026161
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainBagDt )
0.9792145951780864
```

? Compare the accuracy to the decision tree accuracy in test.

```
>>> sklearn.metrics.balanced_accuracy_score( y_test,  predTestDt )
0.6018565458892587
>>> sklearn.metrics.balanced_accuracy_score( y_test,  predTestBagDt )
0.6427002727637082
```

Better. But not great.

Random Forests

Like bagged trees but...

- Sample variables at each split.
- Sampling on columns, not observations.
- Helps understand variable importance.
- Also averages across many trees.

? Set up a pipeline using `RandomForestClassifier` from the `sklearn.ensemble` package.

```
>>> from sklearn.ensemble import RandomForestClassifier
> pipeRf = Pipeline([
      ('preprocessor', preprocessor ),
      ('model',        RandomForestClassifier() )
  ], verbose=True )
```

Fit the random forest model on the training data.

```
>>> pipeRf.fit( X_train, y_train )
```

Use the fitted model to predict in training and test.

```
>>> predTrainRf = pipeRf.predict( X_train )
>>> predTestRf = pipeRf.predict( X_test )
```

❓ Compare the accuracy to the prior tree accuracy in training.

```
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainDt )
0.997556867026161
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainBagDt )
0.9792145951780864
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainRf )
0.9959862161264482
```

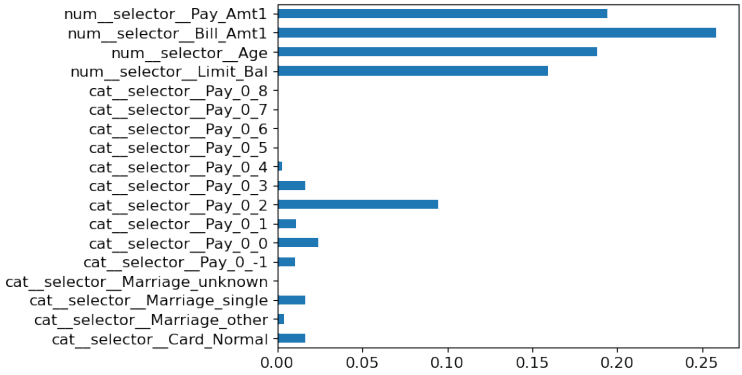
❓ Compare the accuracy to the prior tree accuracy in test.

```
>>> sklearn.metrics.balanced_accuracy_score( y_test, predTestDt )
0.6018565458892587
>>> sklearn.metrics.balanced_accuracy_score( y_test, predTestBagDt )
0.6427002727637082
>>> sklearn.metrics.balanced_accuracy_score( y_test, predTestRf )
0.6474992367390477
```

A benefit of sampling on columns is that we can figure out which columns are important.

? How important is each variable to the random forest model?

```
>>> rfBar = pandas.Series( pipeRf[ "model" ].feature_importances_,  index=pipeRf[:-1].
    get_feature_names_out() ).plot.barh()
```



Boosted Trees

Like others trees but...

- Sequential, not parallel.
- Builds from weak learners (e.g., places the model didn't do well in a prior iteration).

? Set up a pipeline using `AdaBoostClassifier` from the `sklearn.ensemble` package.

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> pipeAda = Pipeline([
        ('preprocessor', preprocessor ),
        ('model',        AdaBoostClassifier() )
    ], verbose=True )
```

Fit the AdaBoost model on the training data.

```
>>> pipeAda.fit( X_train, y_train )
```

Use the fitted model to predict in training and test.

```
>>> predTrainAda = pipeAda.predict( X_train )
>>> predTestAda = pipeAda.predict( X_test )
```

Compare the accuracy to the prior tree accuracy in training.

```
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainDt )
0.997556867026161
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainBagDt )
0.9792145951780864
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainRf )
0.9959862161264482
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainAda)
0.6484697067385655
```

Compare the accuracy to the prior tree accuracy in test.

```
>>> sklearn.metrics.balanced_accuracy_score( y_test, predTestDt )
0.6018565458892587
>>> sklearn.metrics.balanced_accuracy_score( y_test, predTestBagDt )
0.6427002727637082
>>> sklearn.metrics.balanced_accuracy_score( y_test, predTestRf )
0.6474992367390477
>>> sklearn.metrics.balanced_accuracy_score( y_test,predTestAda )
0.6451736465781409
```

? Set up a pipeline using `GradientBoostingClassifier` from the `sklearn.ensemble` package.

```
>>> from sklearn.ensemble import GradientBoostingClassifier
> pipeGbA = Pipeline([
      ('preprocessor', preprocessor ),
      ('model',        GradientBoostingClassifier() )
  ], verbose=True )
```

Fit the GradientBoost model on the training data.

```
>>> pipeGbA.fit( X_train, y_train )
```

Use the fitted model to predict in training and test.

```
>>> predTrainGbA = pipeGbA.predict( X_train )
>>> predTestGbA = pipeGbA.predict( X_test )
```

? Compare the accuracy to the prior tree accuracy in training.

```
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainDt )
0.997556867026161
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainBagDt )
0.9792145951780864
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainRf )
0.9959862161264482
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainAda)
0.6484697067385655
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainGbA )
0.6515328575894339
```

? Compare the accuracy to the prior tree accuracy in test.

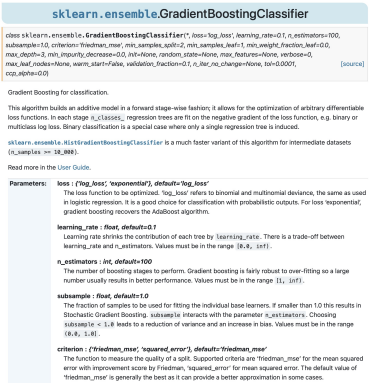
```
>>> sklearn.metrics.balanced_accuracy_score( y_test, predTestDt )
0.6018565458892587
>>> sklearn.metrics.balanced_accuracy_score( y_test, predTestBagDt )
0.6427002727637082
>>> sklearn.metrics.balanced_accuracy_score( y_test, predTestRf )
0.6474992367390477
>>> sklearn.metrics.balanced_accuracy_score( y_test, predTestAda )
0.6451736465781409
```

Hyperparameters

- For linear regression, you don't have many options. Just the model.
- But ML models often have multiple parameters to tune them.
- These are **hyperparameters**.
- Unlike model weights, you control these and can adjust.
- We've (mostly) taken the defaults so far. How do you decide if the defaults are best?

Help on GradientBoostingClassifier

Check help: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html`.



sklearn.ensemble.GradientBoostingClassifier

class sklearn.ensemble.**GradientBoostingClassifier**(*, loss='log_loss', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0*) [source]

Gradient Boosting for classification.

This algorithm builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage `n_classes_` regression trees are fit on the negative gradient of the loss function, e.g. binary or multiclass log loss. Binary classification is a special case where only a single regression tree is induced.

`sklearn.ensemble.HistGradientBoostingClassifier` is a much faster variant of this algorithm for intermediate datasets (`n_samples >= 10_000`).

Read more in the User Guide.

Parameters:

loss : {'log_loss', 'exponential'}, default='log_loss'
The loss function to be optimized. 'log_loss' refers to binomial and multinomial deviance, the same as used in logistic regression. It is a good choice for classification with probabilistic outputs. For loss 'exponential', gradient boosting recovers the AdaBoost algorithm.

learning_rate : float, default=0.1
Learning rate shrinks the contribution of each tree by `learning_rate`. There is a trade-off between learning_rate and n_estimators. Values must be in the range `[0.0, inf)`.

n_estimators : int, default=100
The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance. Values must be in the range `[1, inf)`.

subsample : float, default=1.0
The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. `subsample` interacts with the parameter `n_estimators`. Choosing `subsample < 1.0` leads to a reduction of variance and an increase in bias. Values must be in the range `(0.0, 1.0]`.

criterion : {'friedman_mse', 'squared_error'}, default='friedman_mse'
The function to measure the quality of a split. Supported criteria are 'friedman_mse' for the mean squared error with improvement score by Friedman, 'squared_error' for mean squared error. The default value of 'friedman_mse' is generally the best as it can provide a better approximation in some cases.

What is all that????

Tuning GradientBoostingClassifier

? Set up another pipeline using `GradientBoostingClassifier` but change the `learning_rate` parameter to 0.6.

```
>>> pipeGbB = Pipeline([
        ('preprocessor', preprocessor ),
        ('model',        GradientBoostingClassifier( learning_rate = 0.6 ) )
    ], verbose=True )
```

❓ Fit the model on the training data.

```
>>> pipeGbB.fit( X_train, y_train )
```

❓ Use the fitted model to predict in training and test.

```
>>> predTrainGbB = pipeGbB.predict( X_train )
>>> predTestGbB = pipeGbB.predict( X_test )
```

❓ Compare the accuracy to the prior tree accuracy in training and test.

```
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainGbA)
0.6515328575894339
>>> sklearn.metrics.balanced_accuracy_score( y_train, predTrainGbB)
0.6946083782701288
```

```
>>> sklearn.metrics.balanced_accuracy_score( y_test,  predTestGbA)
0.6434179893382181
>>> sklearn.metrics.balanced_accuracy_score( y_test,  predTestGbB )
0.6402308251887422
```

We've used the same model (`GradientBoostingClassifier`) but the changing the hyperparameter changed the results.

Searching for the best values for hyperparameters

How do we know...

- What values of the hyperparameters would improve the model? (And wouldn't it be tedious to keep writing this code to try new hyperparameters?)
- How do we know any improvements are likely "real" and not just random chance?

We can...

- Define a search space, a **grid**, of hyperparameter combinations we'd like to try.
- Repeatedly train models on subsets of the training data, **crossfolds**.

A combinatorially large grid search space

The default of `max_depth` is 3. How will results change for values $\{1, 3, 6, 9\}$?

The default of `learning_rate` is 0.1. How will results change for values $\{0, 0.15, 0.3, 0.45, 0.6, 0.75\}$?

		learning_rate					
		0	0.15	0.3	0.45	0.6	0.75
max_depth	1	(1, 0)	(1, 0.15)	(1, 0.3)	(1, 0.45)	(1, 0.6)	(1, 0.75)
	3	(3, 0)	(3, 0.15)	(3, 0.3)	(3, 0.45)	(3, 0.6)	(3, 0.75)
	6	(6, 0)	(6, 0.15)	(6, 0.3)	(6, 0.45)	(6, 0.6)	(6, 0.75)
	9	(9, 0)	(9, 0.15)	(9, 0.3)	(9, 0.45)	(9, 0.6)	(6, 0.75)

6 values of `learning_rate` and 4 values of `max_depth` yields $6 * 4 = 24$ combinations.

Careful

Grid search space can grow quickly and take considerable time to run. If you have a weaker computer, use fewer values for each hyperparameter.

Define the grid of hyperparameters to try by using a Python dictionary.

```
>>> paramGridGbA = {'model__learning_rate': [ 0, 0.15, 0.3, 0.45, 0.6, 0.75 ],
                    'model__max_depth': [1, 3, 6, 9]
                    }
```

Names of the dictionary entries are important:

1. Step name exactly as you specified the name in the pipeline.
2. Two underscores ("__")
3. The parameter name, exactly as the documentation says.

Crossfolds

If we just tried a change to a parameter one time, we wouldn't be sure the change itself made a difference. Changes could be from the stochastic optimization process. So we try a few times and summarize.

For k crossfolds of the training data...

1. Randomly shuffle the dataset
2. Split the dataset into k groups.
3. For each group:
    1. Group is a *test* subset.
    2. Other groups are *training* data.
    3. Fit the model (training).
    4. Evaluate the model (test).
    5. Save the focal score.
4. Summarize the k scores, including measures of distribution.

Crossfolds illustration

Consider a hypothetical dataset of 6 rows labeled $\{A, B, C, D, E, F\}$.

A `test_size` of $\frac{1}{6}$ could yield training data of $\{A, B, C, E, F\}$ and test data of $\{D\}$.

$k = 5$ crossfolds of the training data would be:

1. Training subset of $\{\quad B, A, E, C\}$; test subset of $\{F\}$
2. Training subset of $\{F, \quad A, E, C\}$; test subset of $\{B\}$
3. Training subset of $\{F, B, \quad E, C\}$; test subset of $\{A\}$
4. Training subset of $\{F, B, A, \quad C\}$; test subset of $\{E\}$
5. Training subset of $\{F, B, A, E \quad\}$; test subset of $\{C\}$

? Use the `GridSearchCV` function in the `sklearn.model_selection` package to search the grid for the best combination of paramters.

```
>>> from sklearn.model_selection import GridSearchCV
>>> gridGbAAccuracy = GridSearchCV( pipeGbA, paramGridGbA, cv=5, n_jobs=-1, verbose=4 )
```

(first) Pipeline to use.

(second) Hyperparameter values to test (dictionary).

cv Tells how many crossfolds to split the training data into.

n_jobs Tells how many processors to use (-1 means all processors)

verbose Will give us more information as it runs.

By default, `GridSearchCV` will use accuracy for classifications and R^2 for regression, but you can specify other metrics.

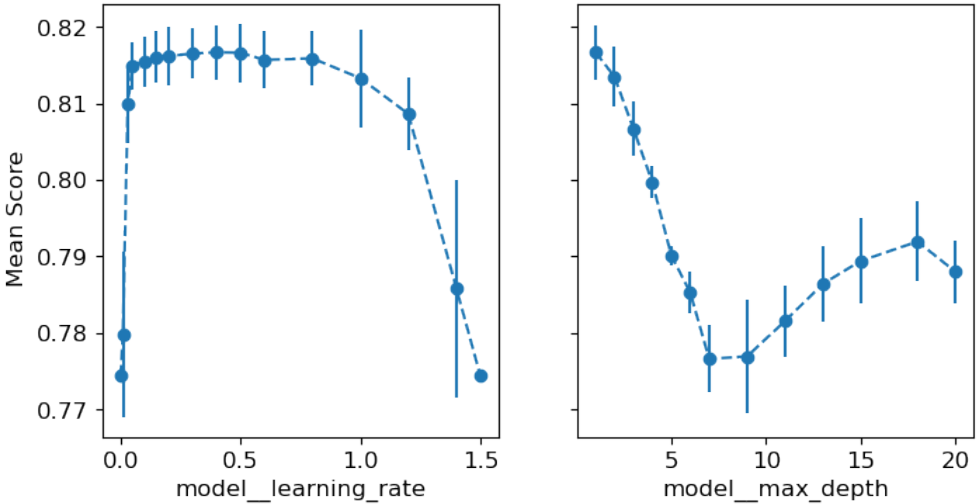
Using a pipeline makes this much much easier than iterating manually.


```
>>> gridGbAAccuracy.fit( X_train, y_train )
```

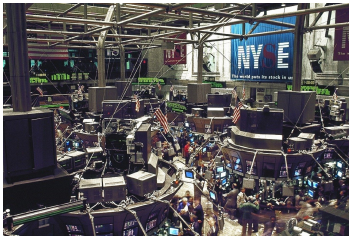
What did the search find?

```
>>> gridGbAAccuracy.best_params_
{'model__learning_rate': 0.3, 'model__max_depth': 1}
>>> gridGbAAccuracy.best_score_
0.8164666666666666
```

Search results: To illustrate, I ran a much larger grid for a few hours.

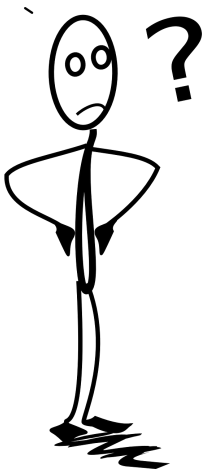


Will the market go up or down?



- Load and profile the "DJIA_table.csv" file.
- Make models that...
  - Set movement as the outcome variable.
  - Include Close, Volume, Open, High, Low as predictors.
  - Normalize all numeric predictors.
  - Extract day of week, month, year from MarketDate and include in your model.
  - Convert all categorical variables except the outcome to dummy variables.
- Build and compare: Logistic, Decision Tree, Bagged Tree, Random Forest, AdaBoost, GradientBoost
- Tune one of the tree models.

Self-Assessment: Can you?



- Prepare
  - Import data? Profile?
- Model
  - Include pipelines for features (categorical and numeric)?
  - Build multiple types of classification models?
- Assess
  - Compare models using multiple criteria and fit criteria to context?
  - Recognize overfitting?
- Tune
  - Select hyperparameters to tune?
  - Use grid search to run hyperparameter variations?
- Debug
  - Selectively add/remove code to isolate problems?
  - Use help and internet to correct?

Today

Learning Objectives

- Build multiple tree models.
- Compare / contrast different tree models.
- Practice models.

Agenda

1. Decision Trees
2. Bagged Trees
3. Random Forests
4. Boosted Trees
5. Tuning

For next class...

- ☐ Complete the Dow Jones example.
- ☐ Next: Deep learning and neural networks.
- ☐ We'll practice more.
- ☐ Email me with any questions or concerns

Thanks!