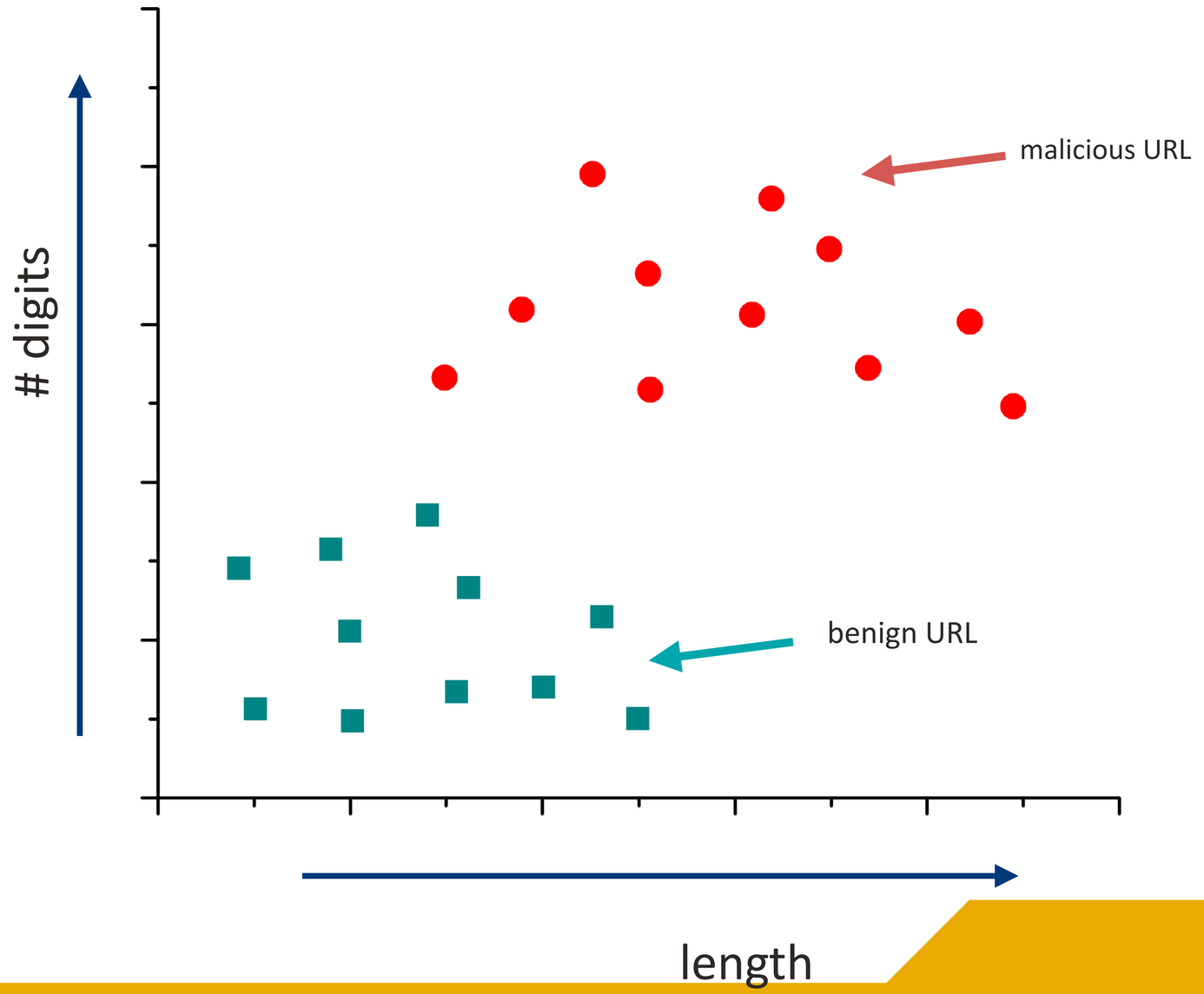
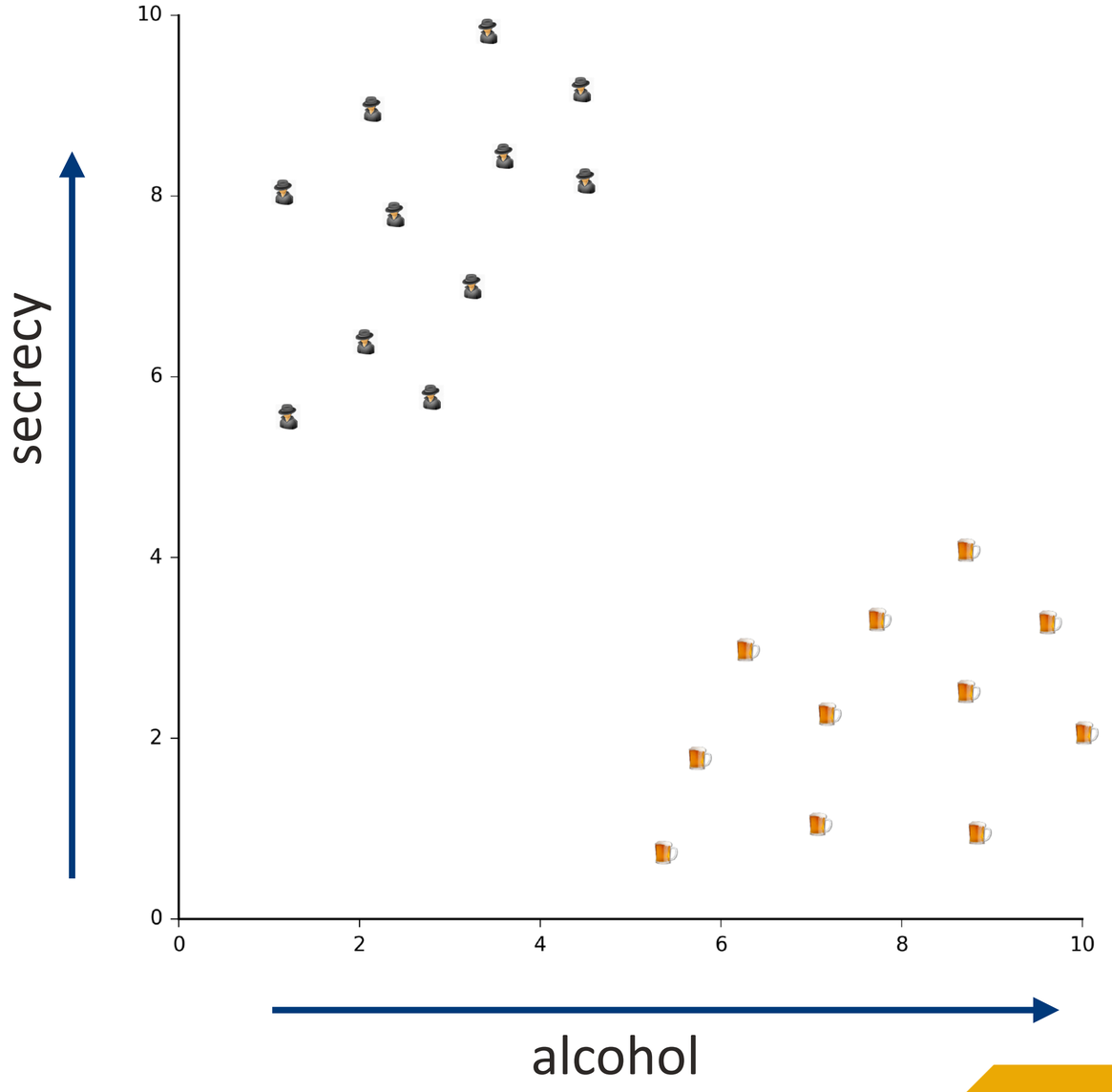
Module 5
Supervised Machine Learning
Classification

Conceptual slides inspired by Sebastian Thrun's https://www.udacity.com/learn/machinelearning

Features and Labels

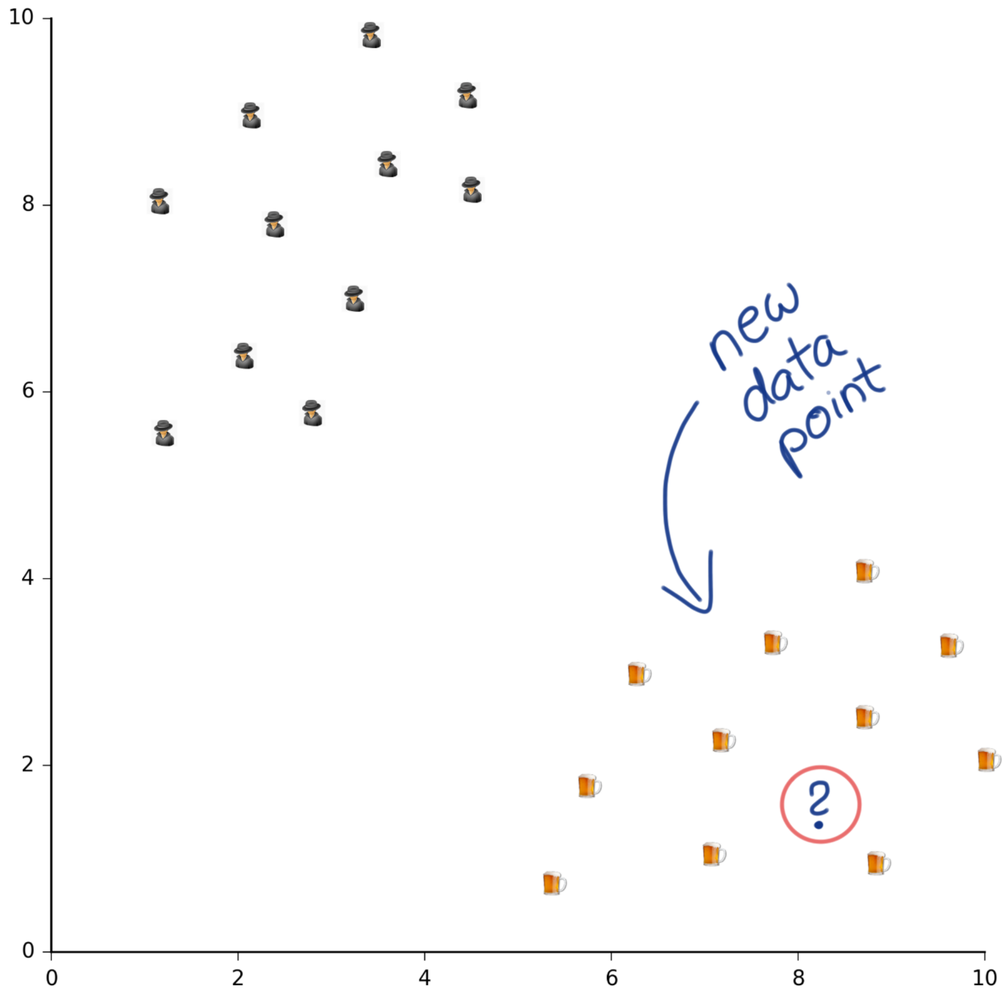


FUN Features and Labels

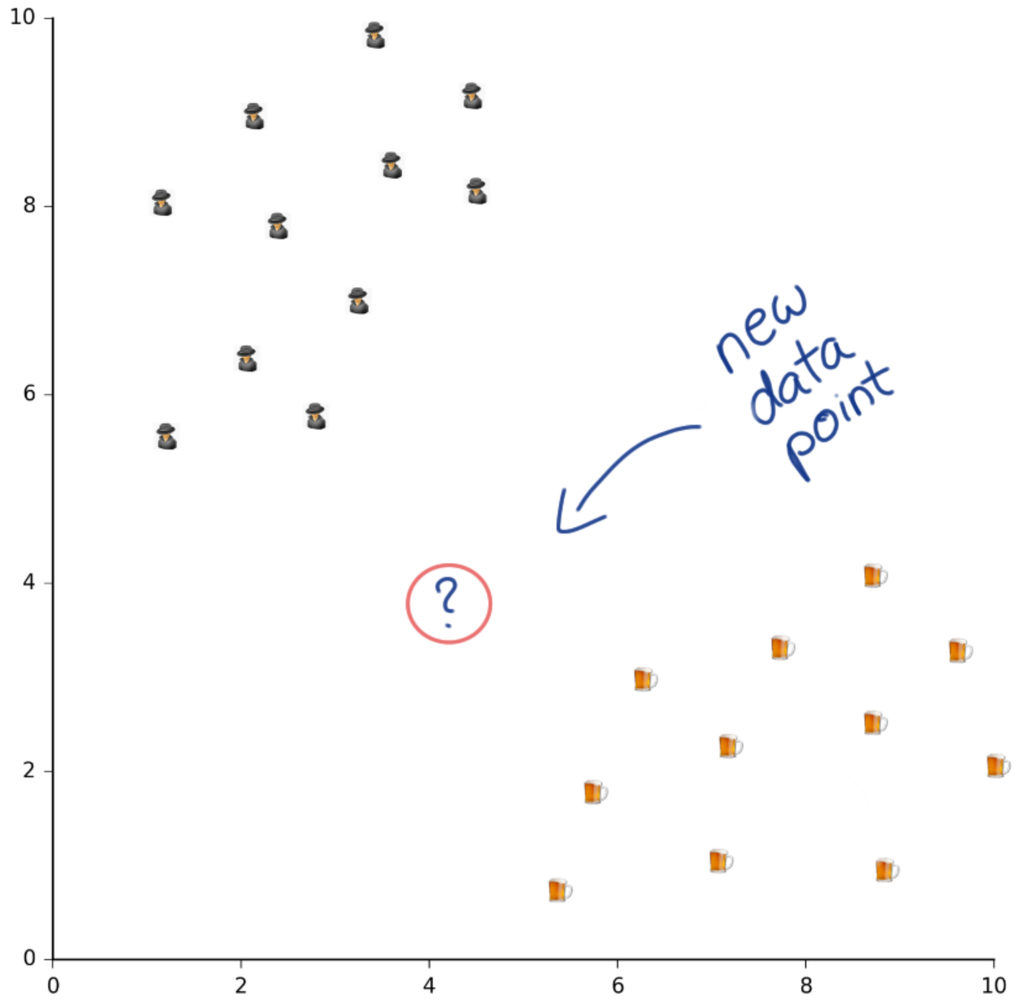


beer and cyber !!!

Making a new decision

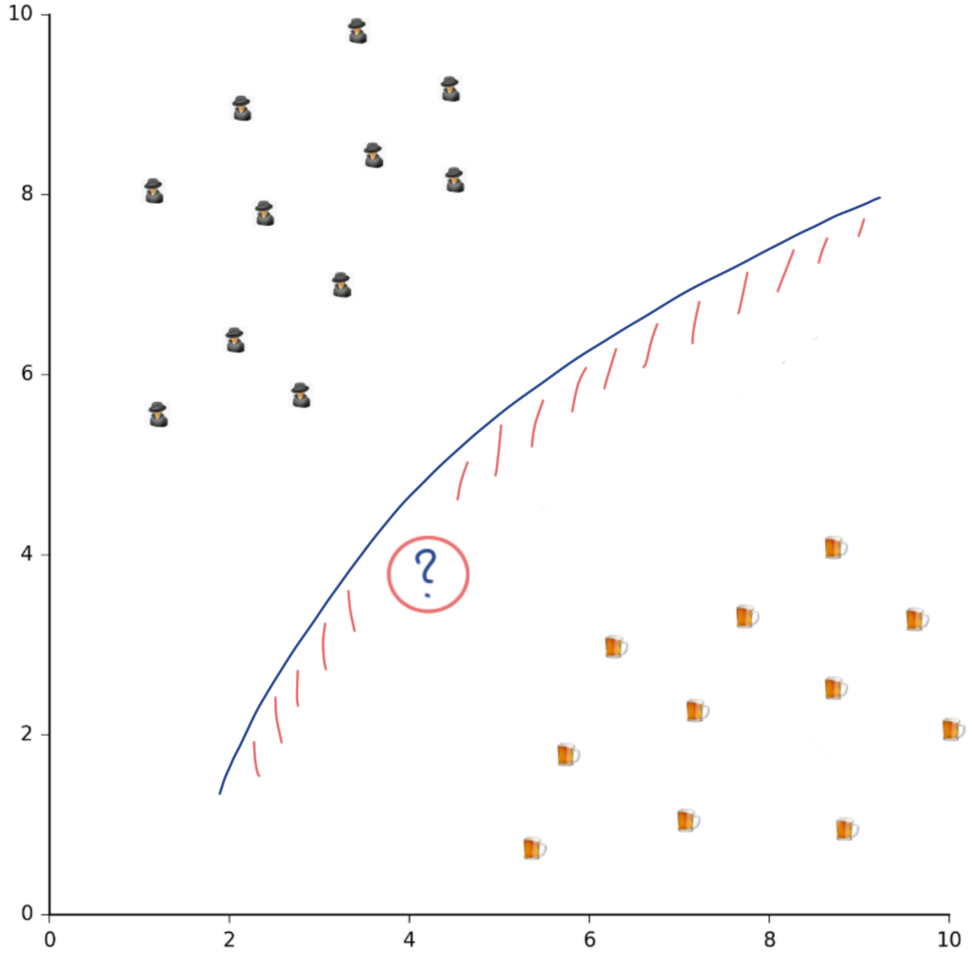


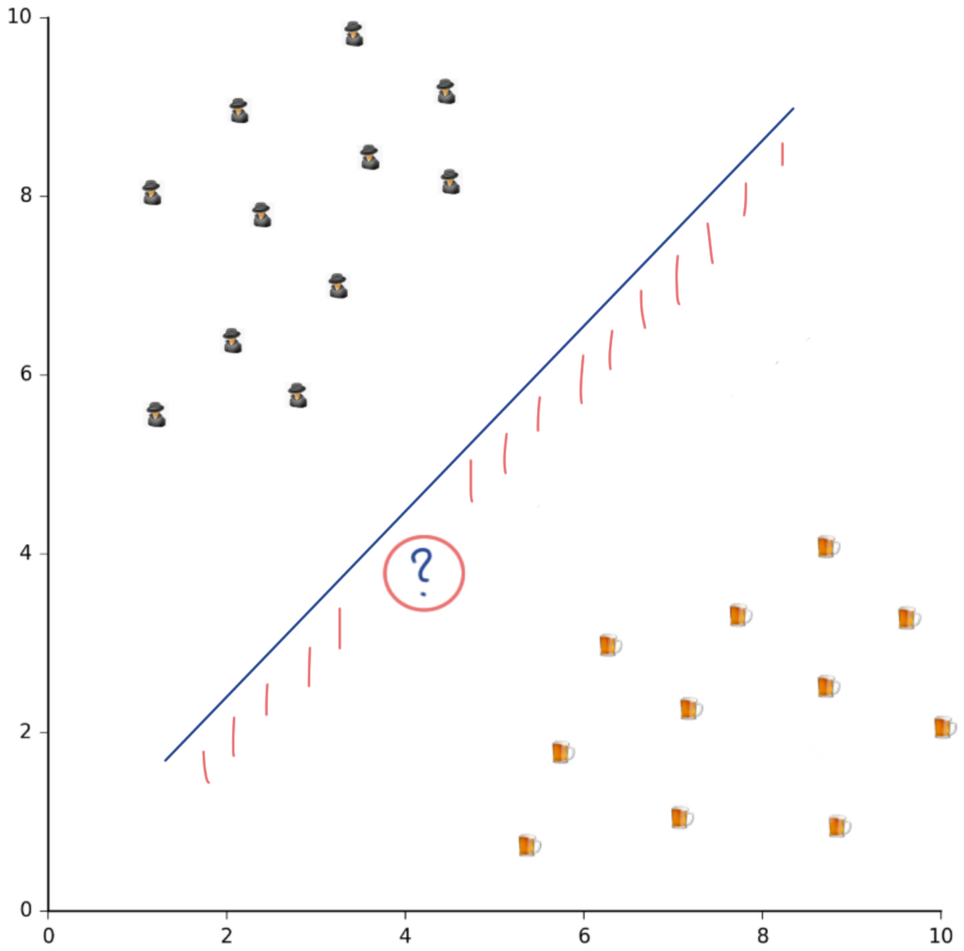
Making a new decision



unclear?

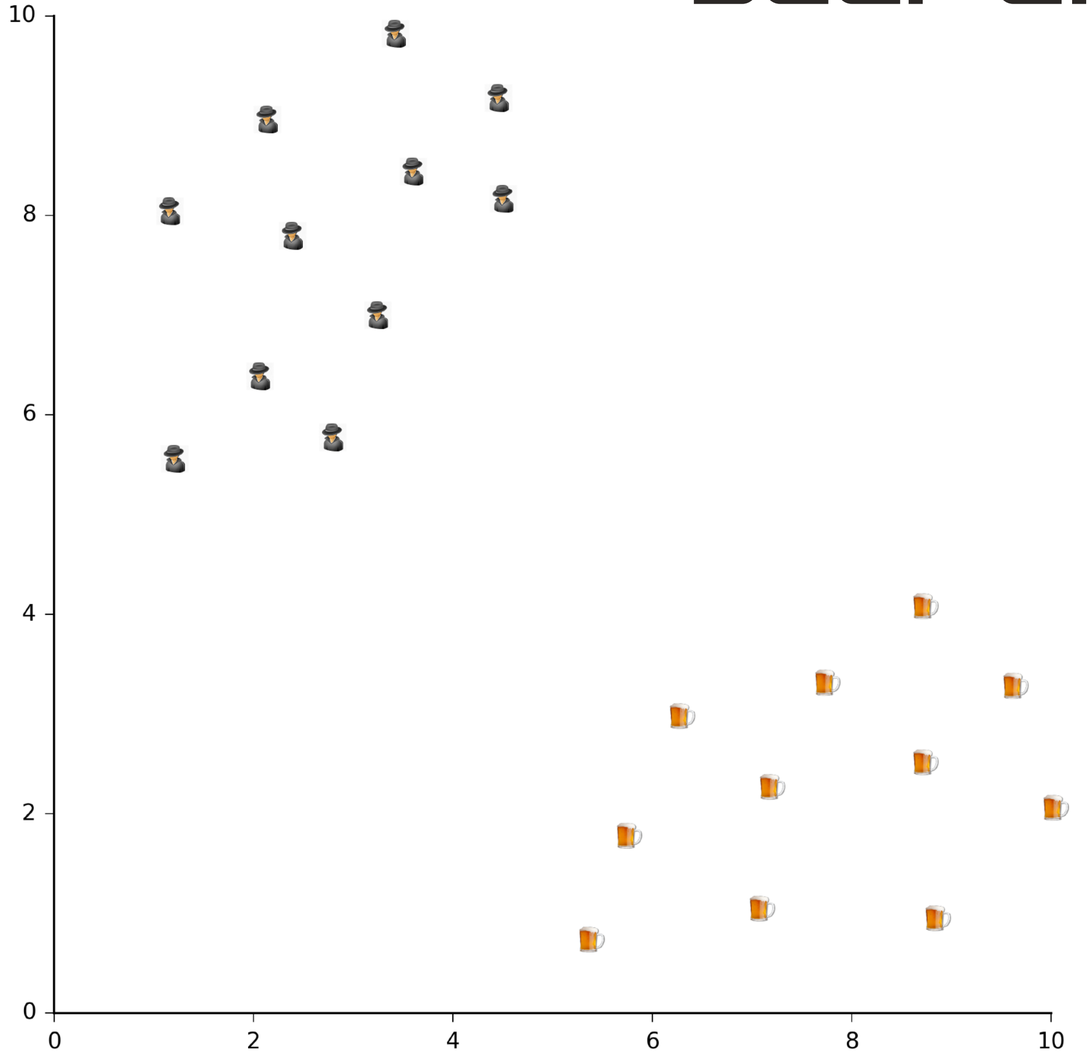
Decision Surface Concept



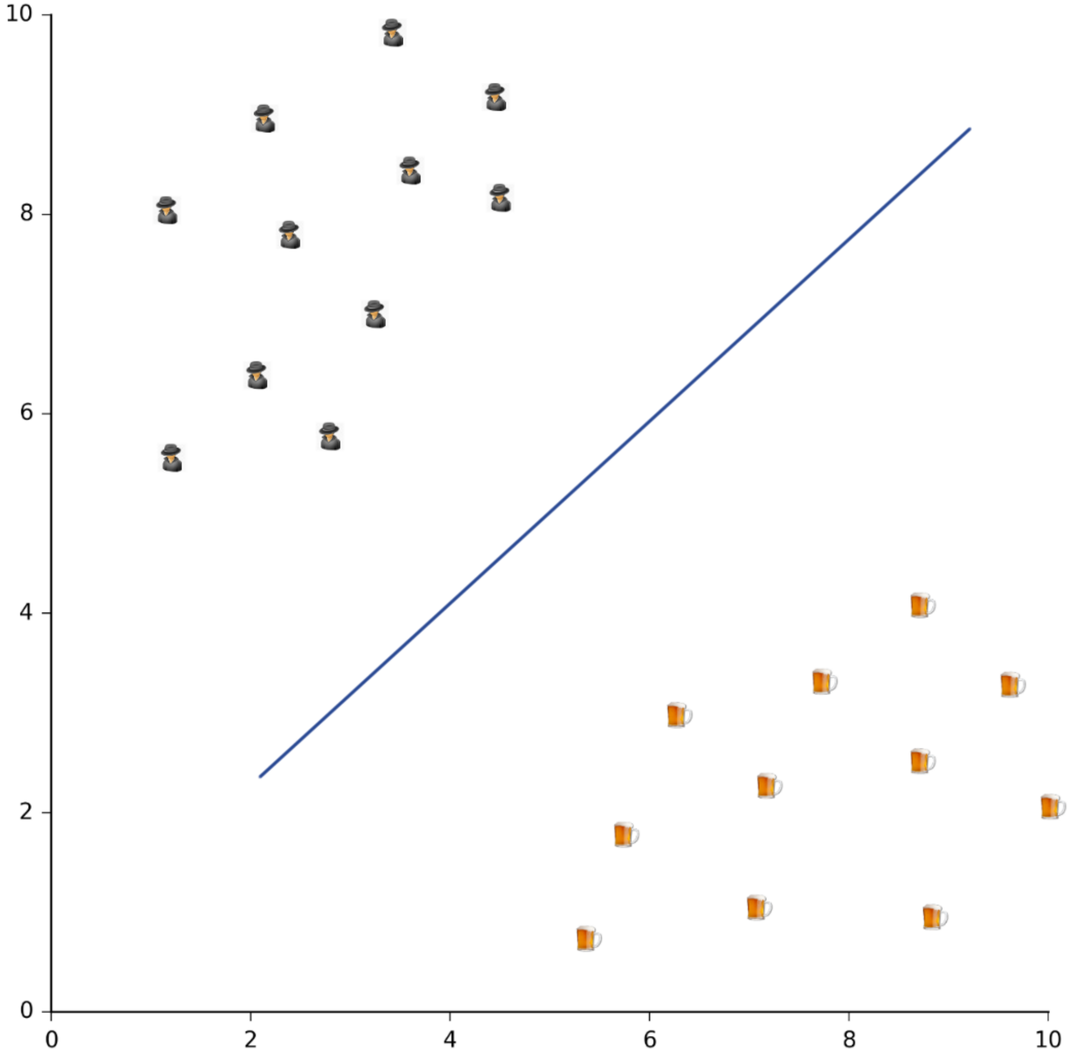


Support Vector Machines (SVM)

How can we linearly separate beer and cyber?

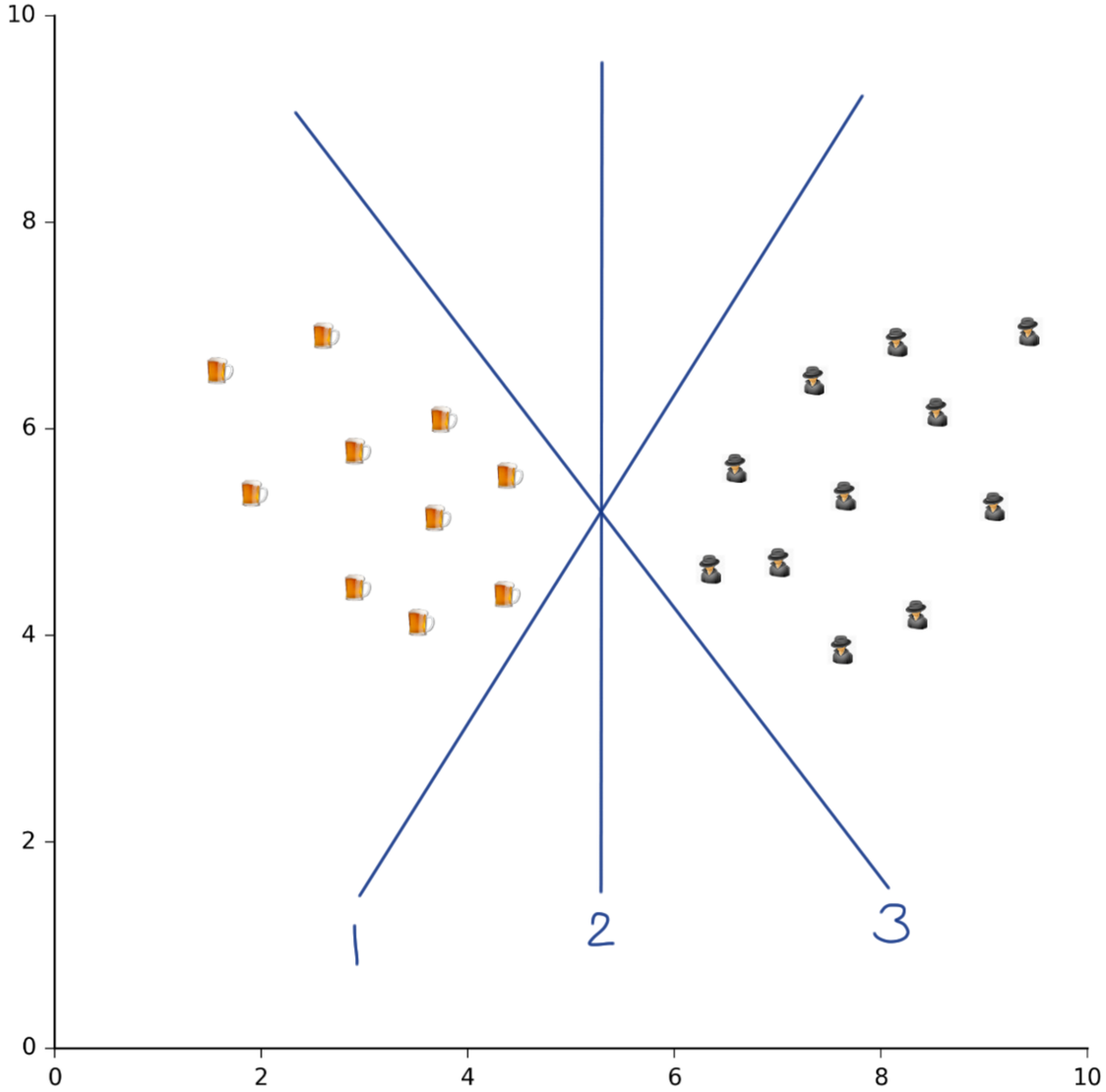


How can we linearly separate beer and cyber?

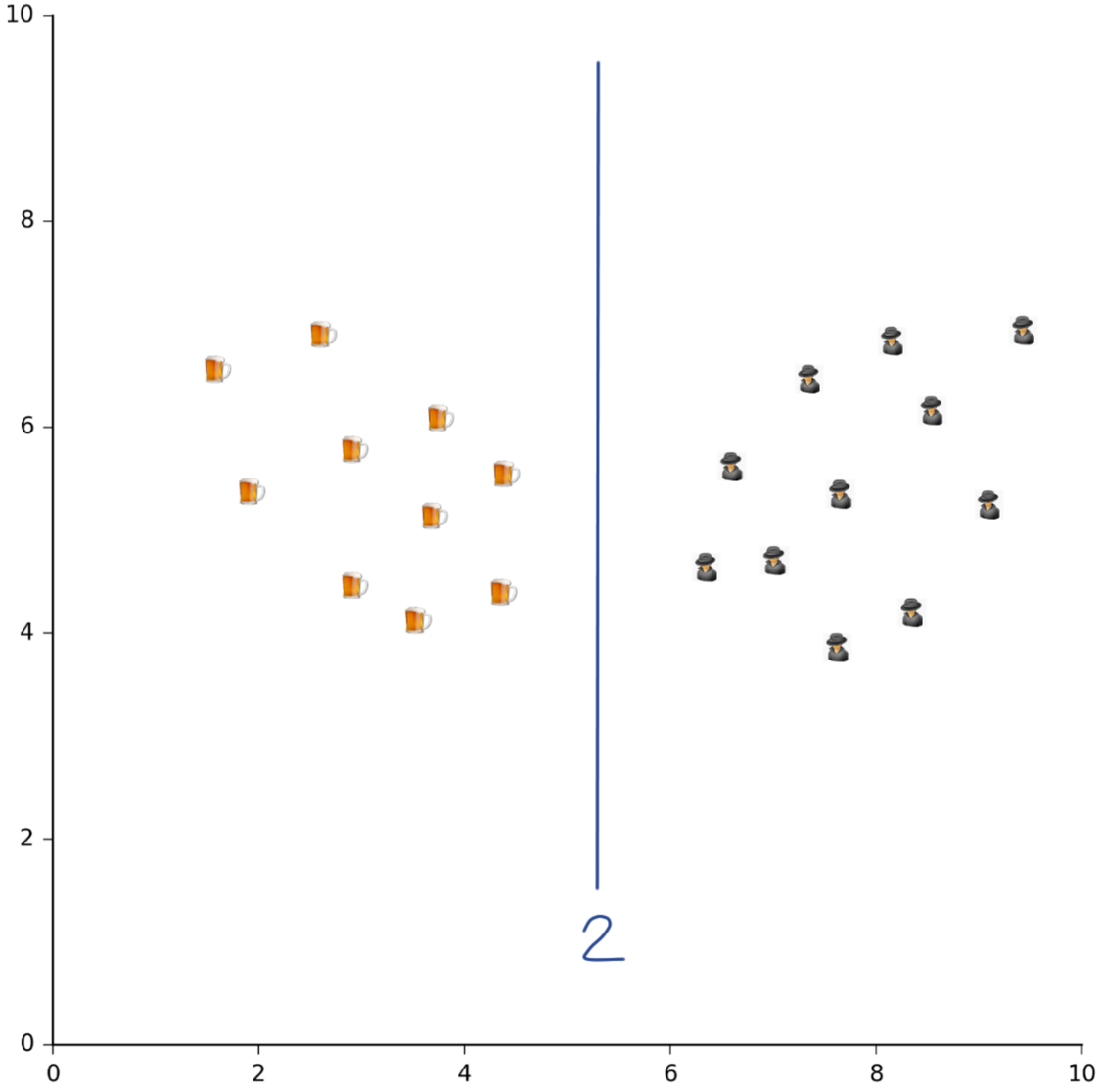


Answer: straight line

Same Question, Choose the Best Line!

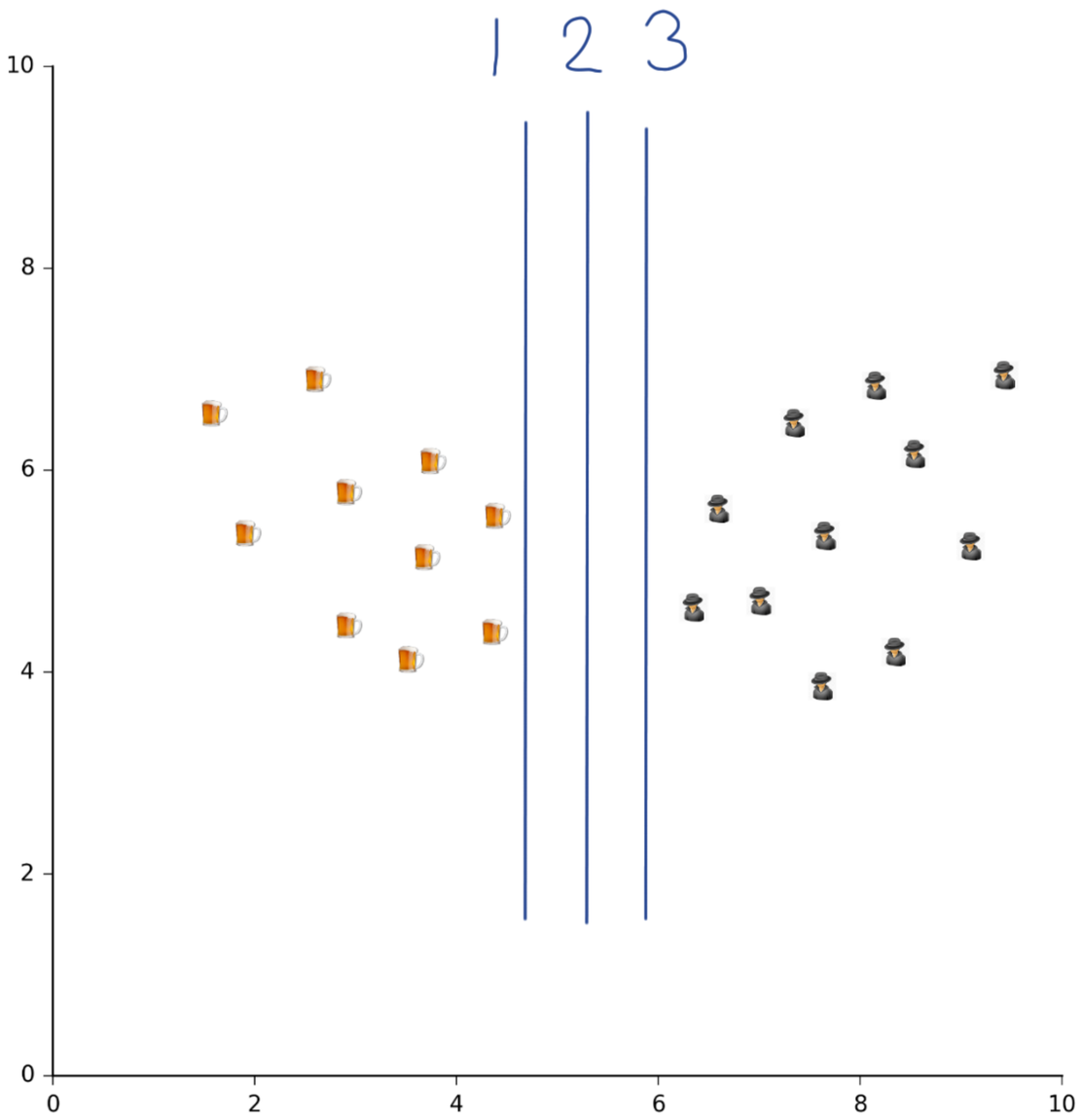


Same Question: Choose the Best Line!

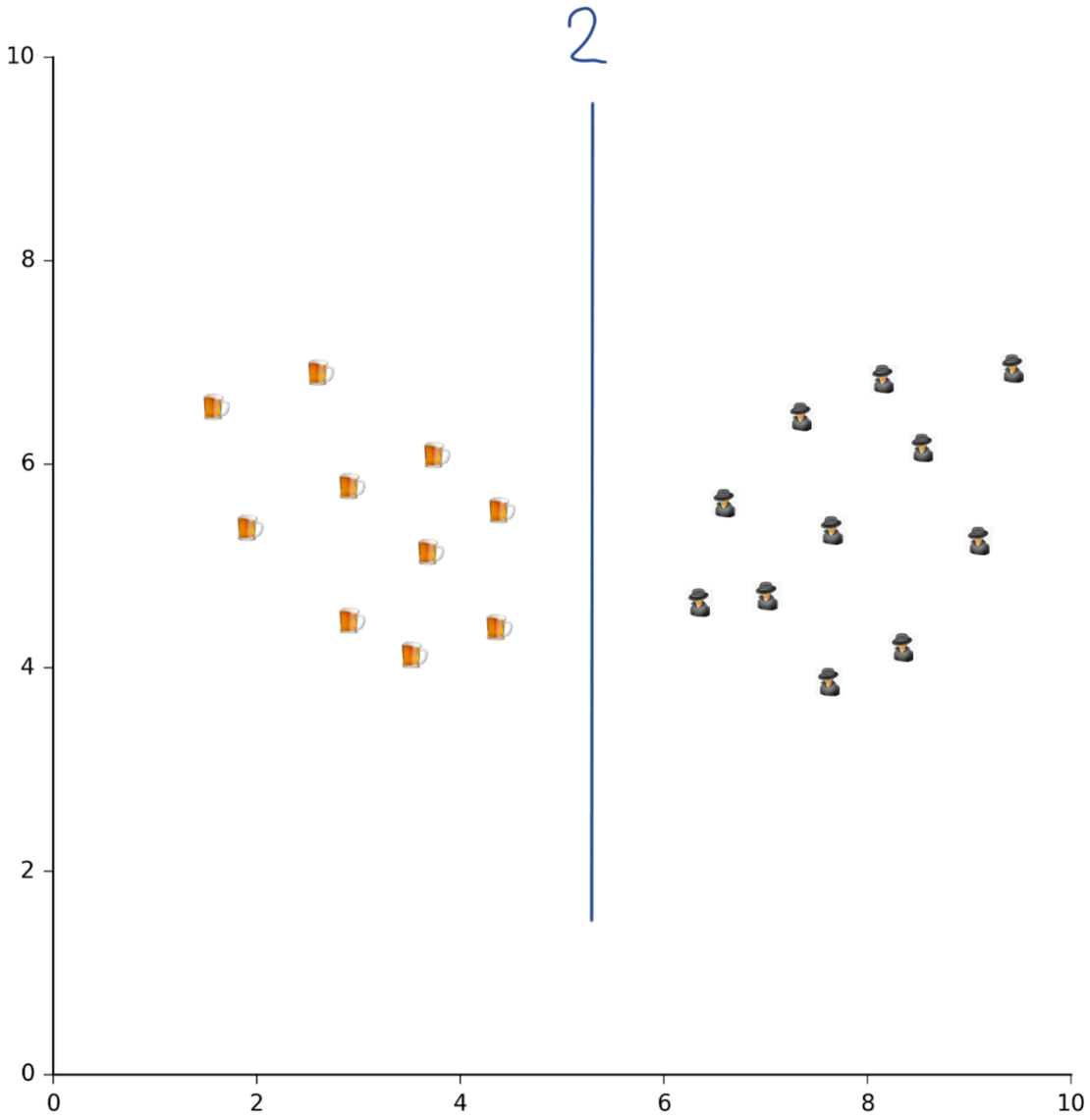


Answer: 2

How about now?

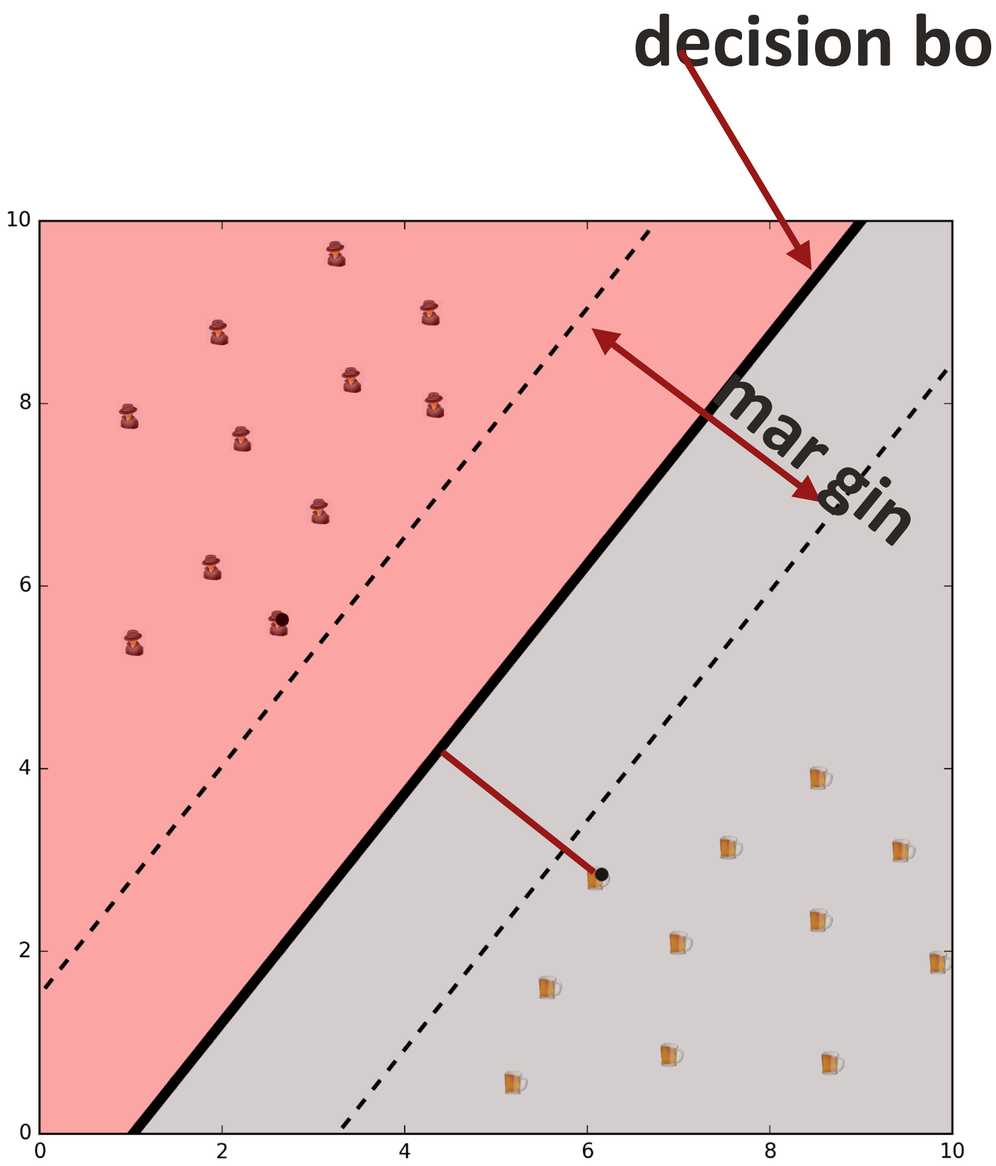


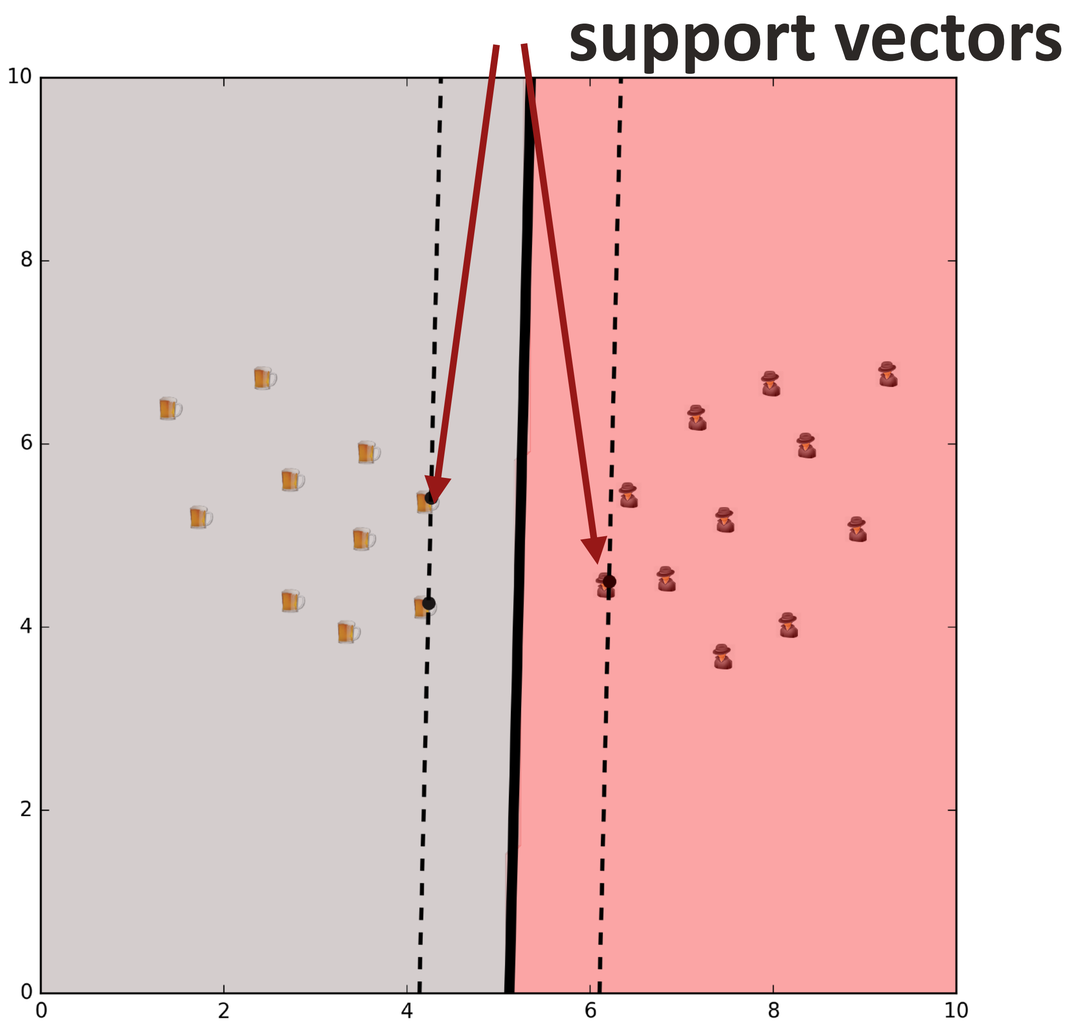
How about now?



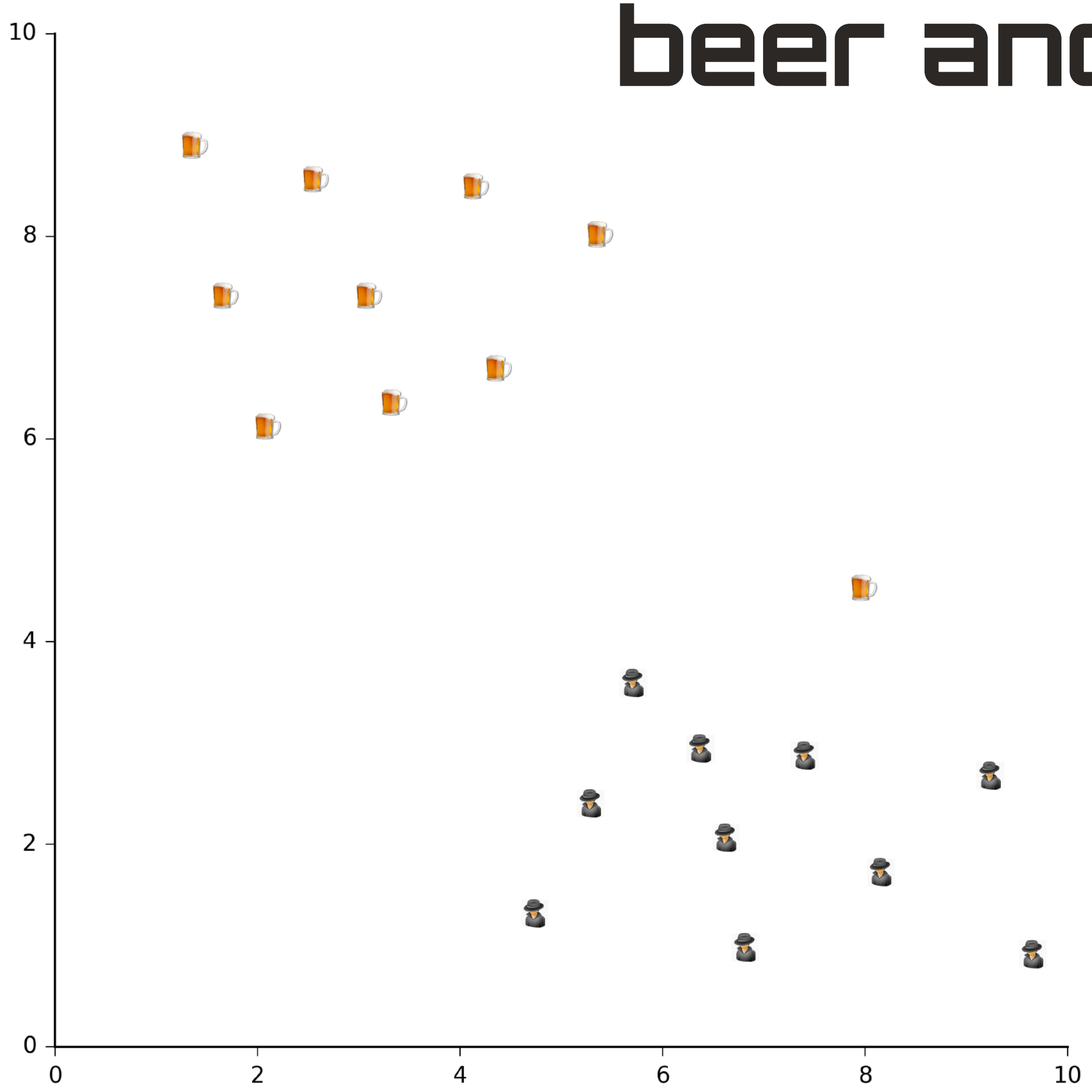
Answer: 2

Open BlackBox...

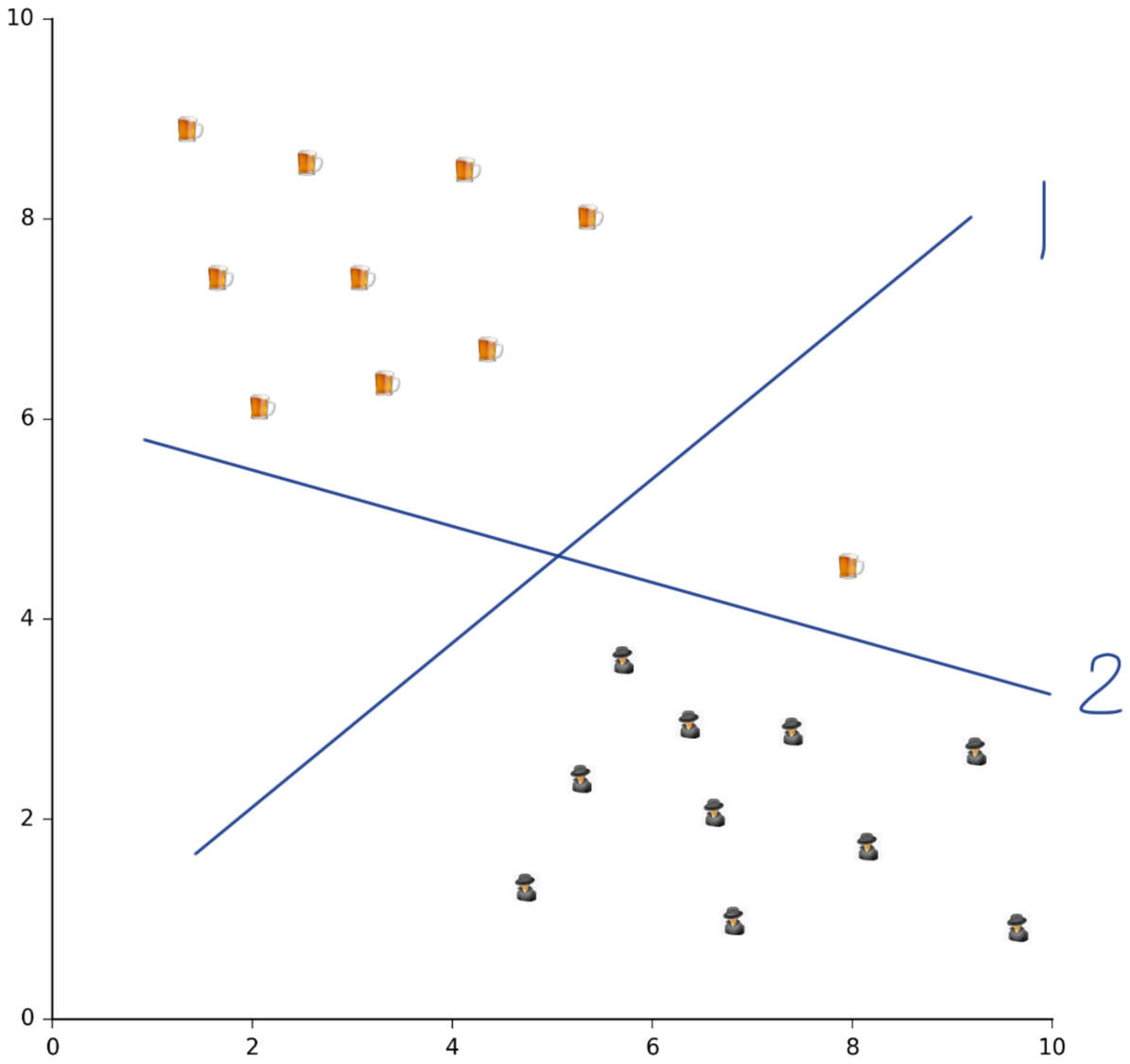




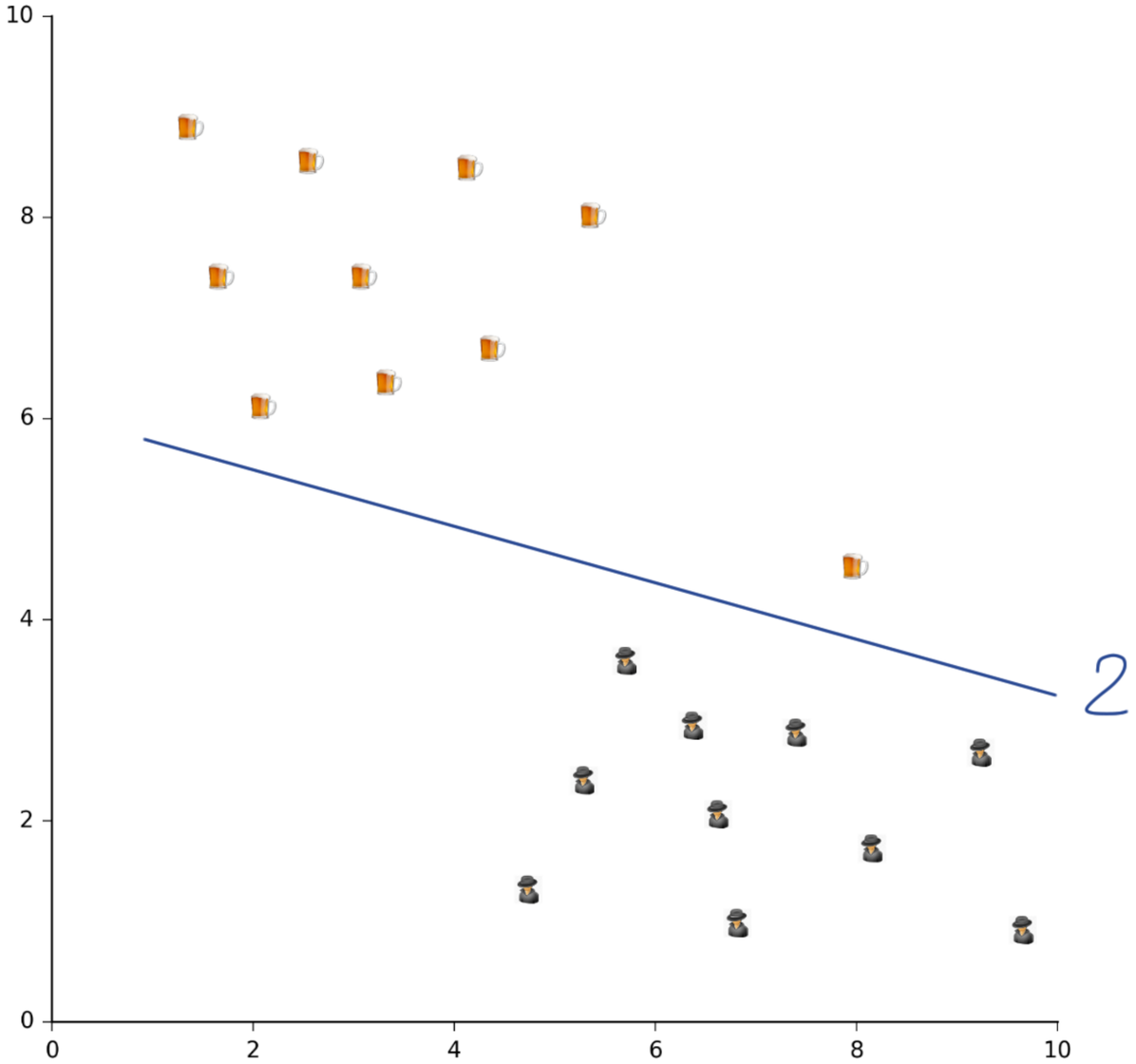
Another Test: How do we best separate beer and cyber linearly?



Choose the Best Line!

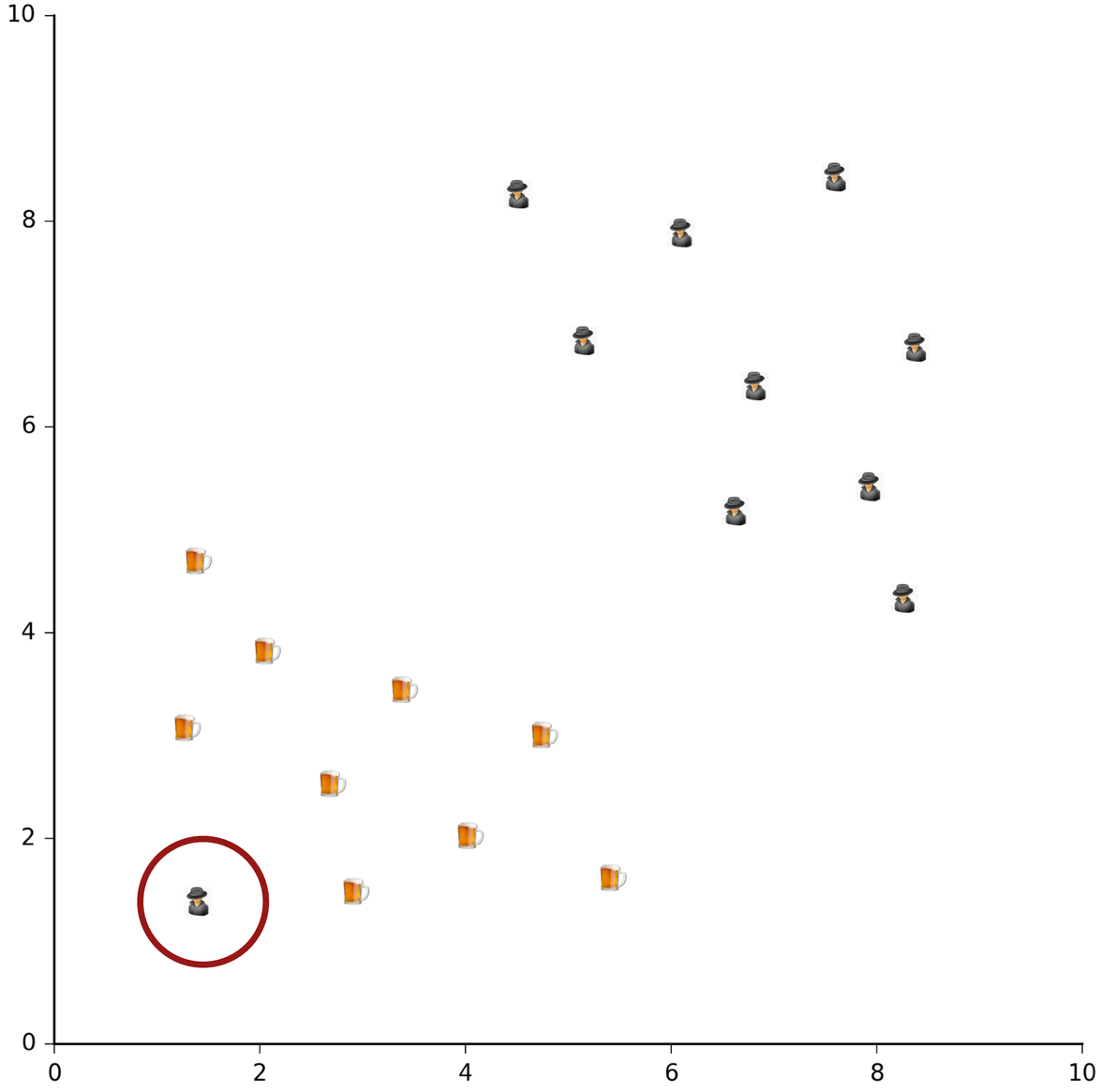


Choose the Best Line!

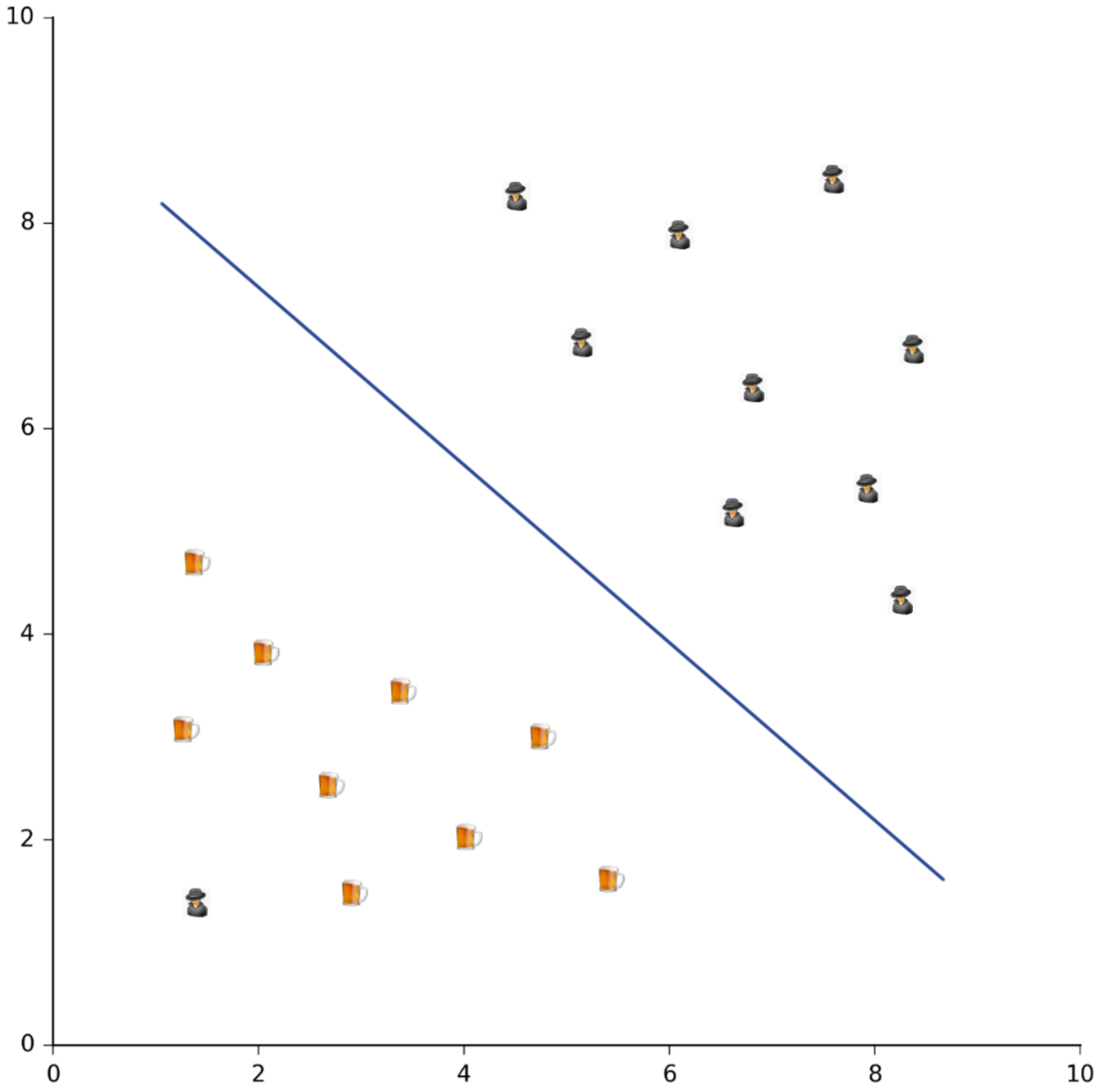


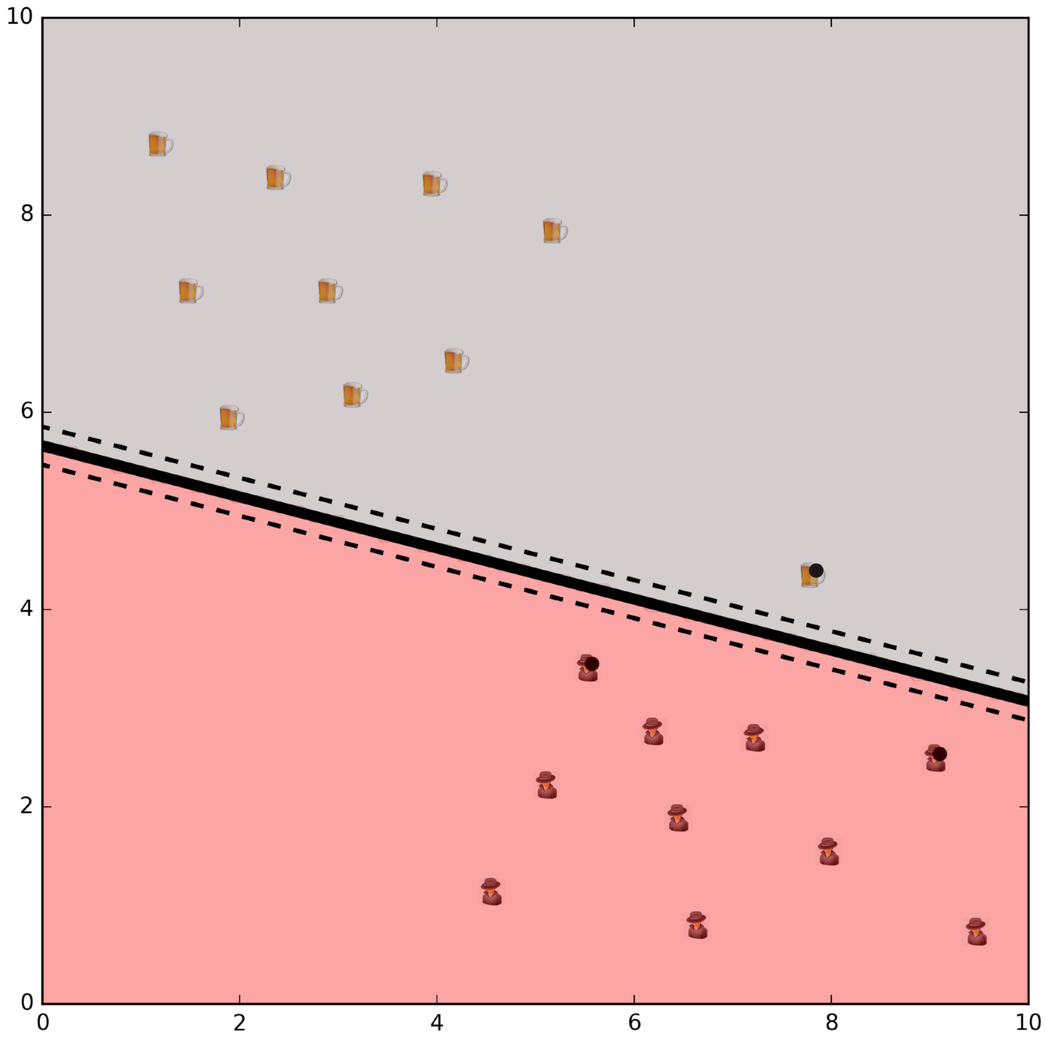
Answer: 2

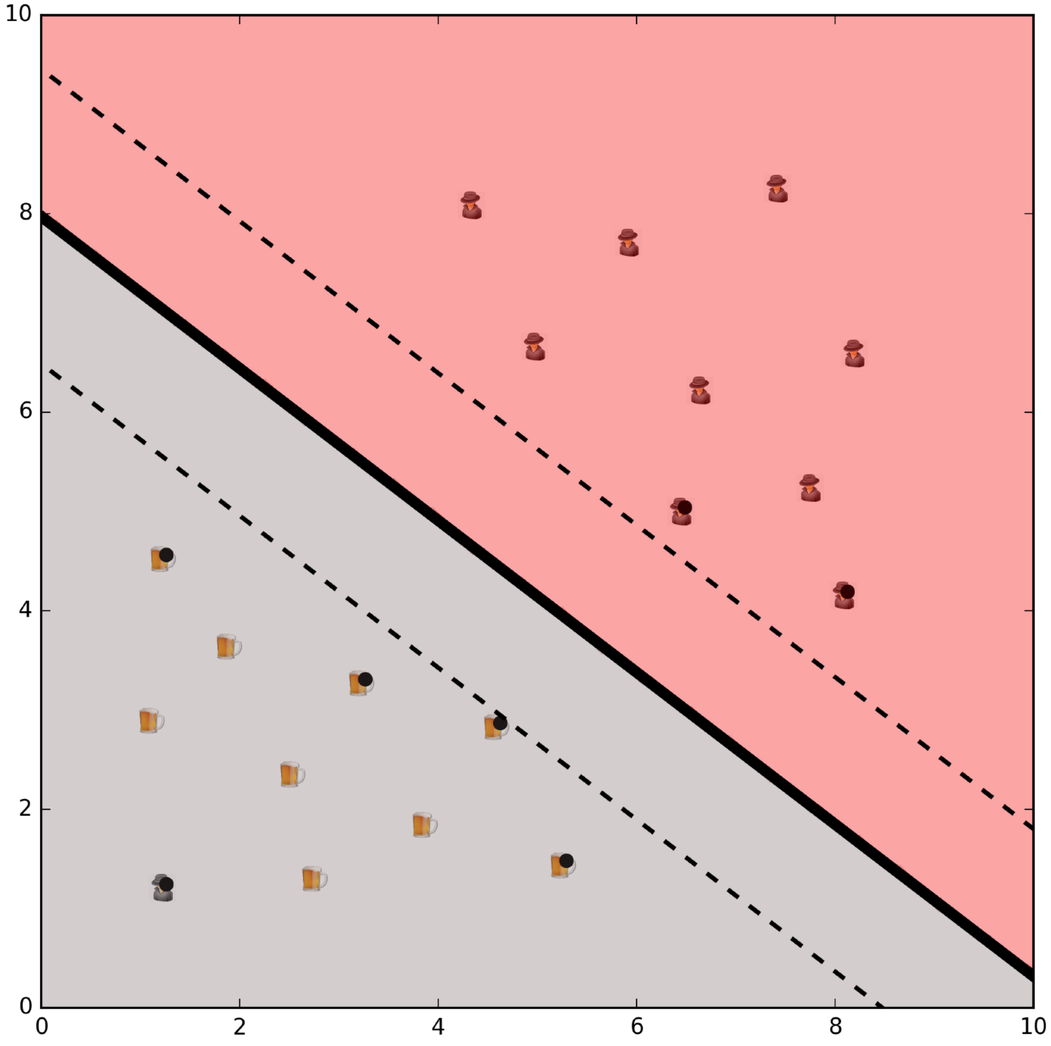
Wait and Now?



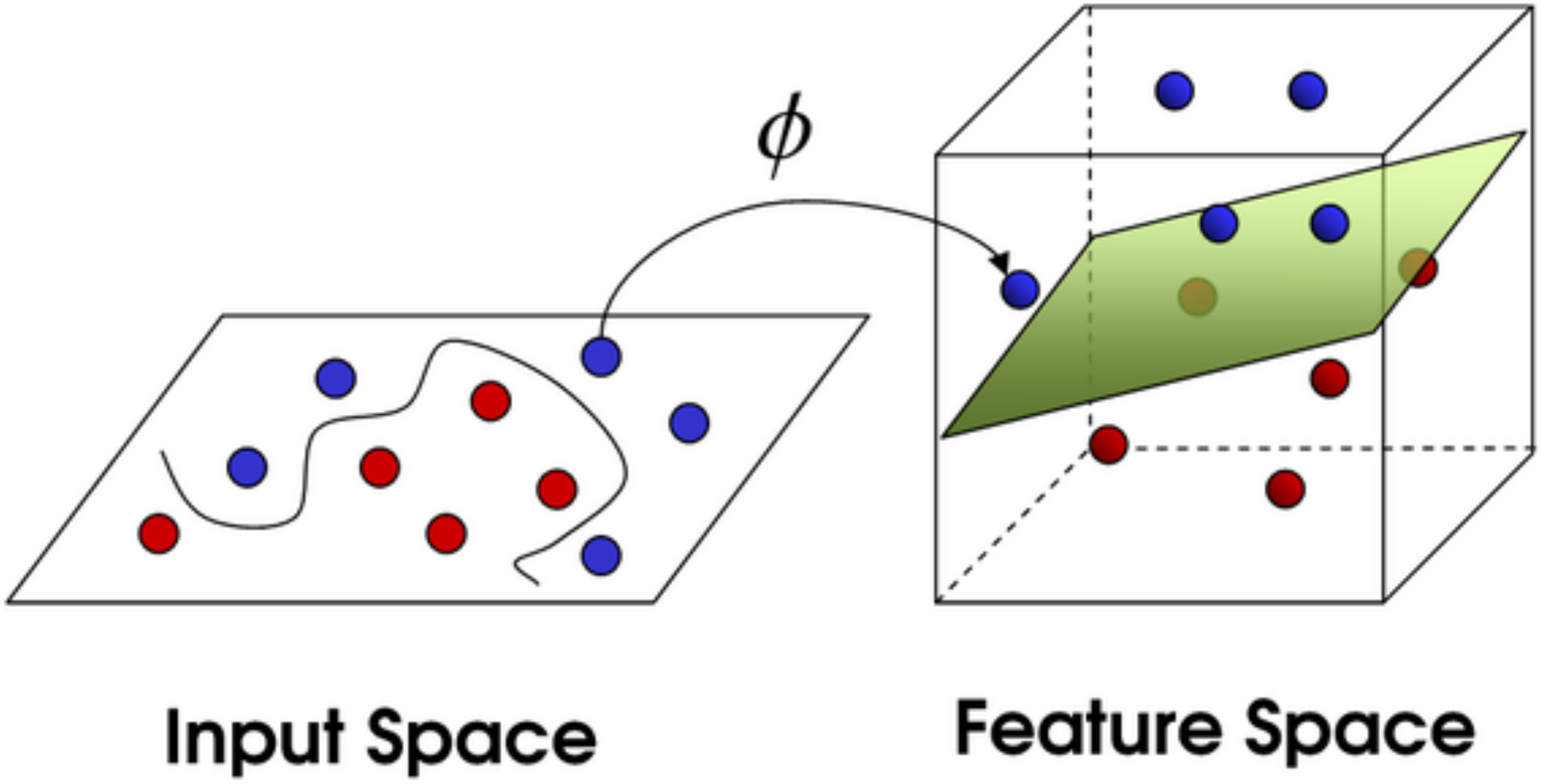
Just ignore?




10
8
6
4
2
0
0
2
4
6
8
10

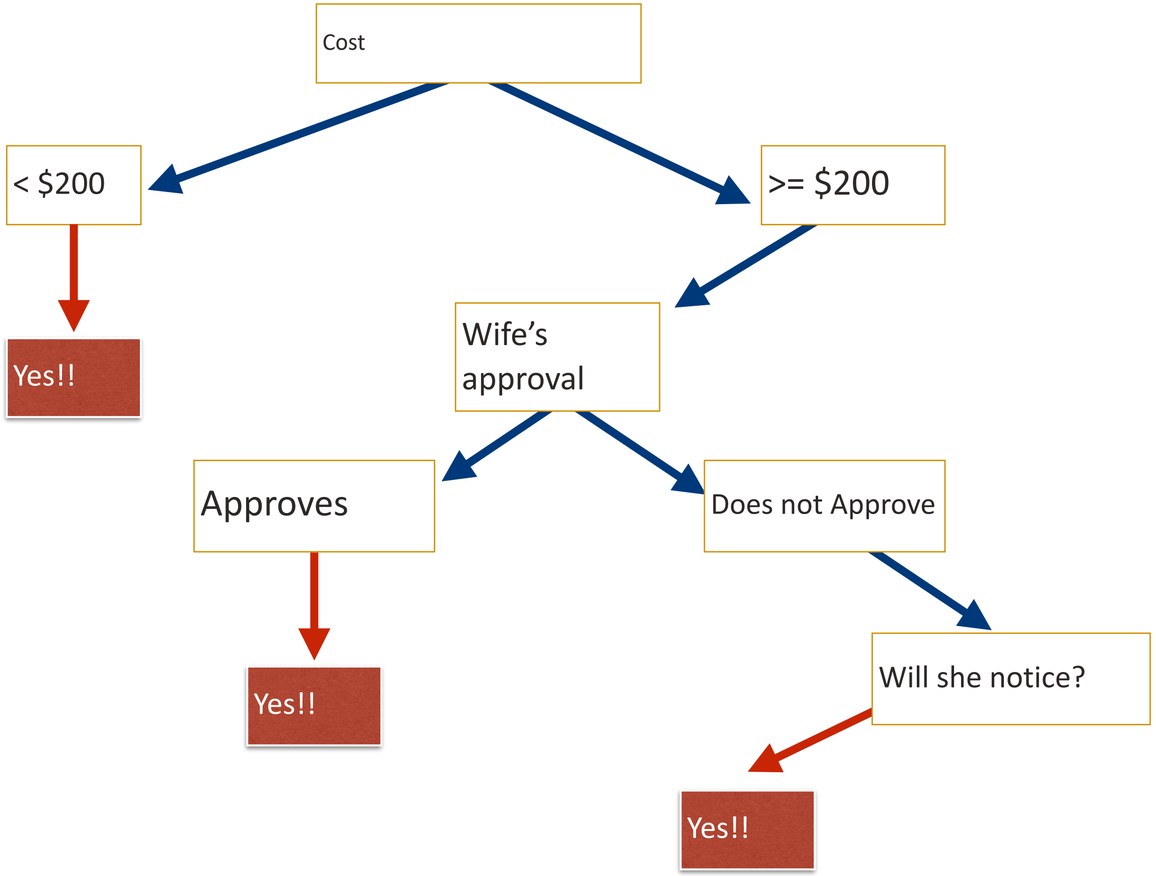

10
8
6
4
2
0
0
2
4
6
8
10

Kernel trick



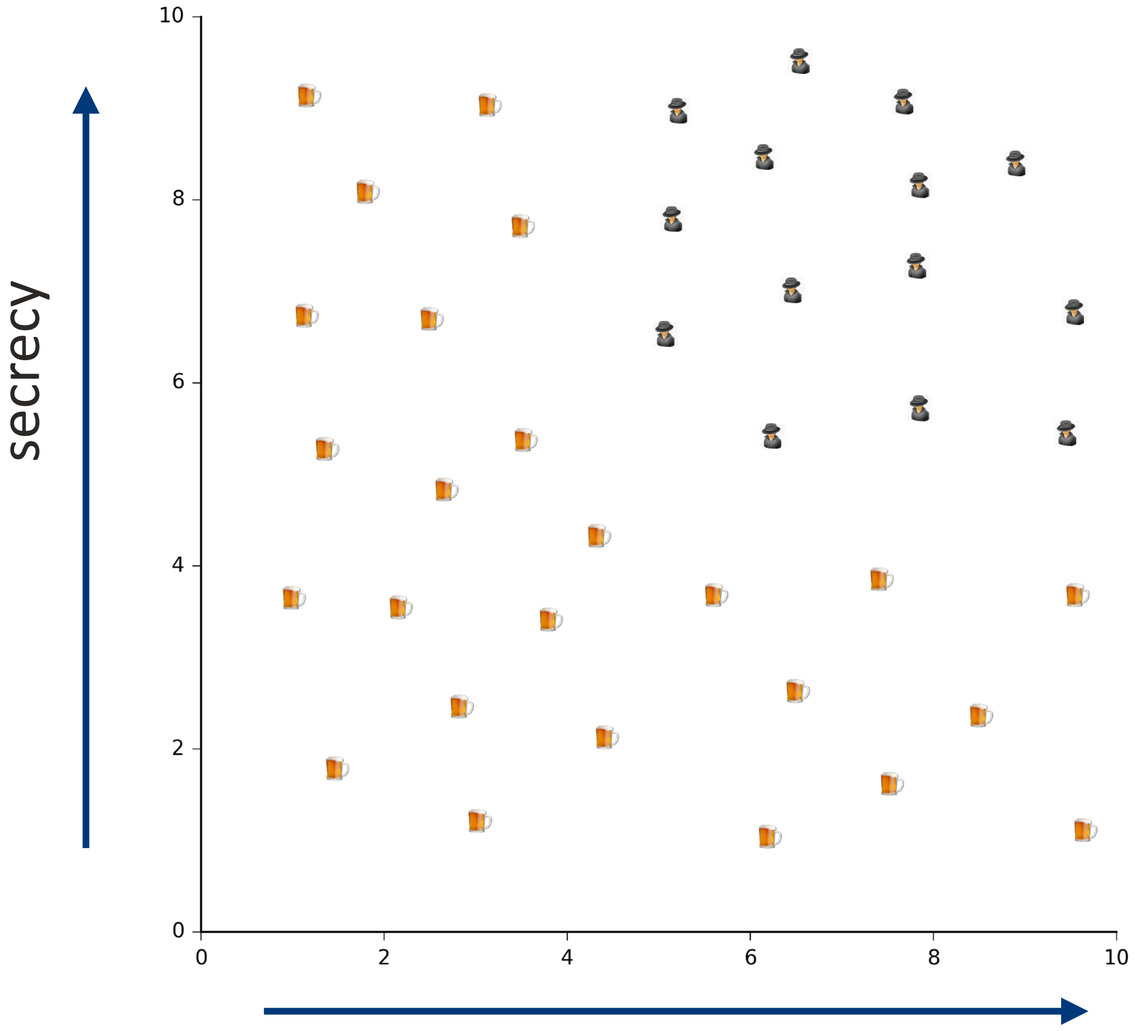
Simple Decision Tree (DT)

Should I buy a new tech gadget?

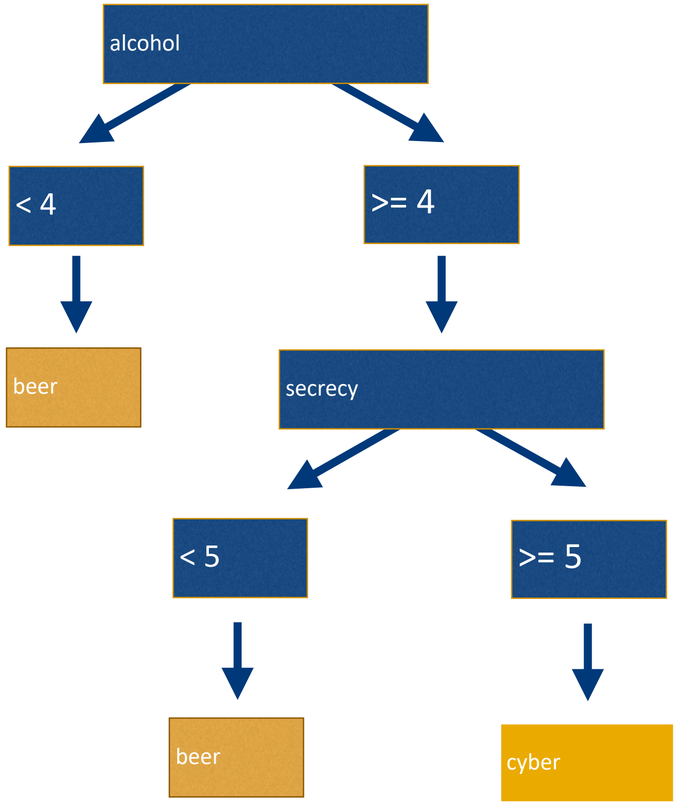


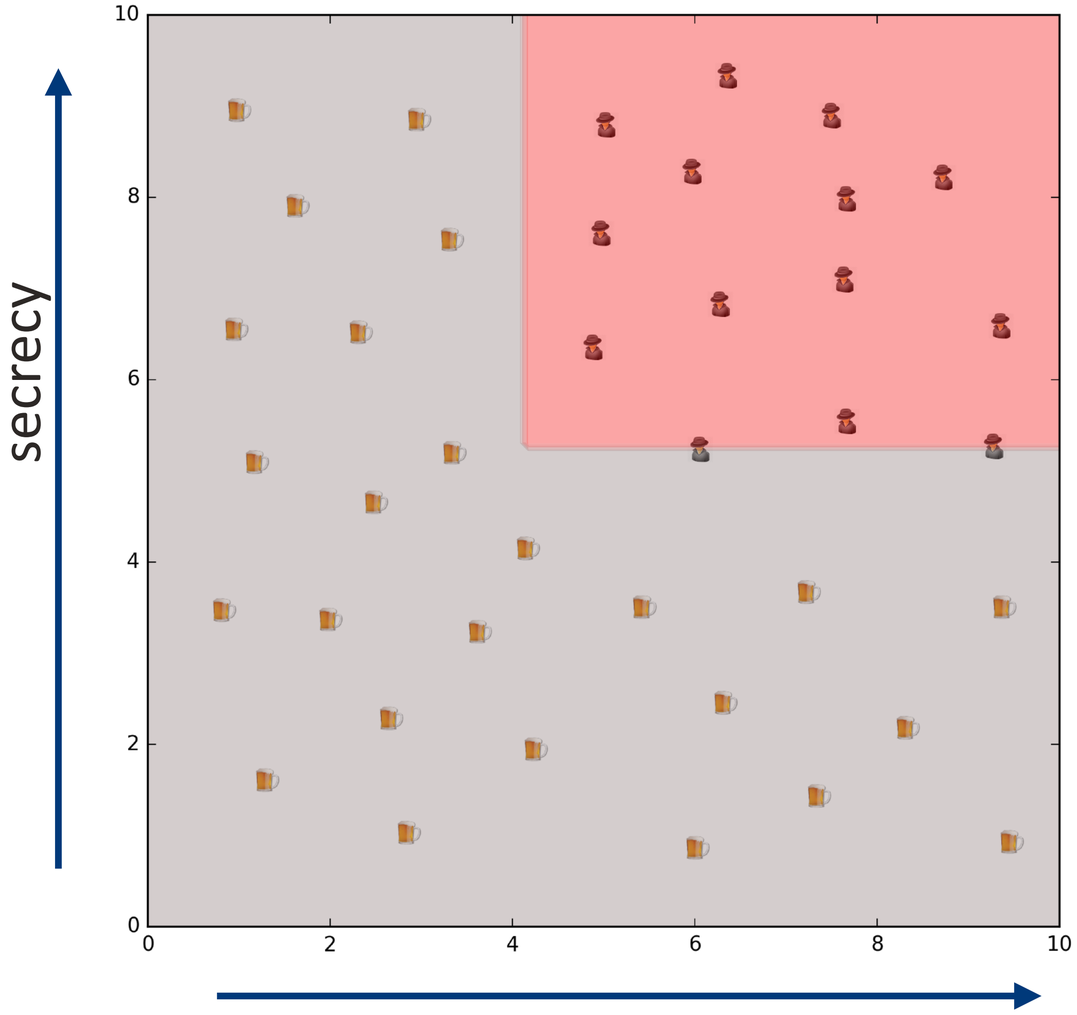
This is Charles' example!

How can we separate beer and cyber?

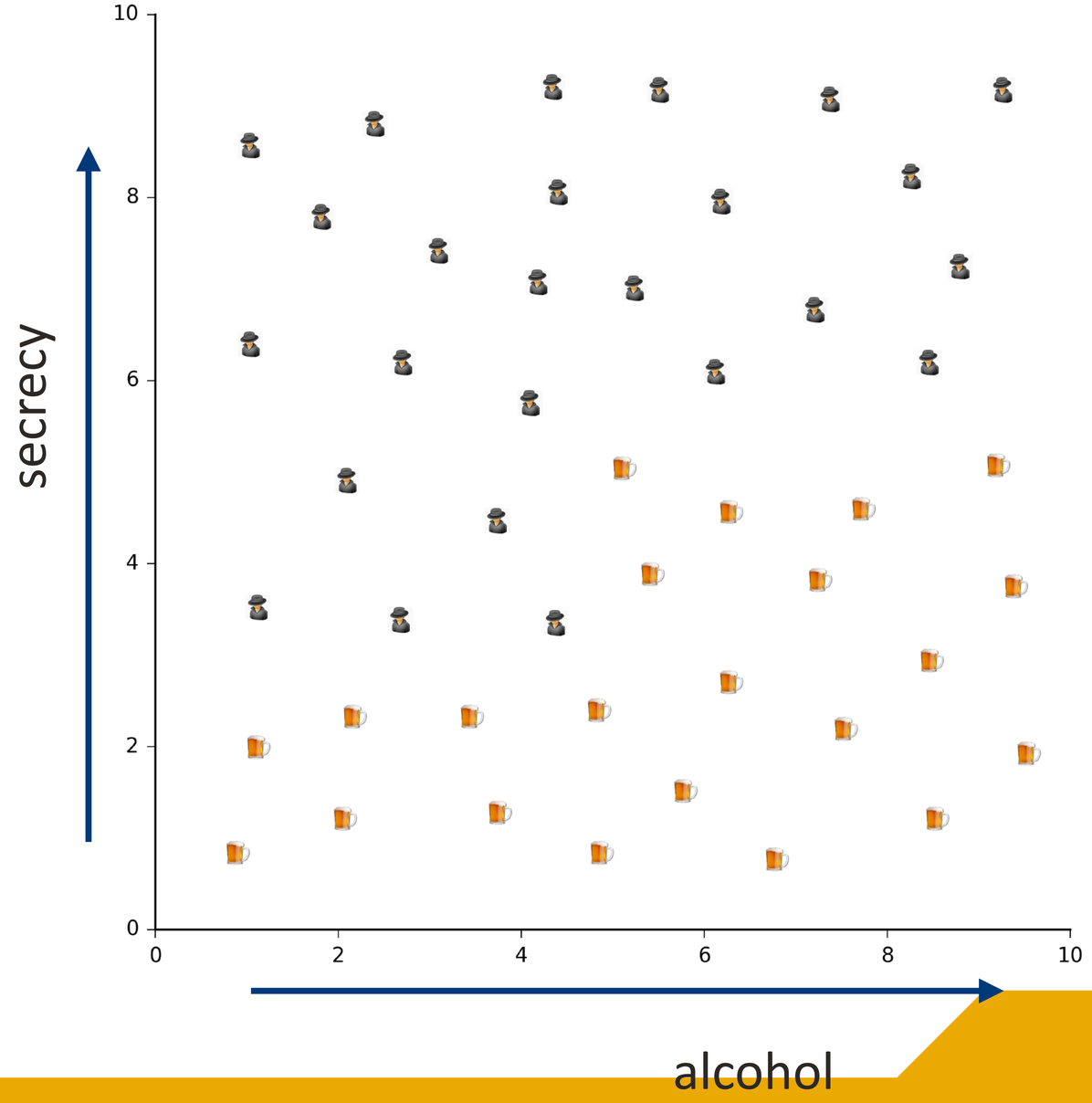


How can we separate beer and cyber?



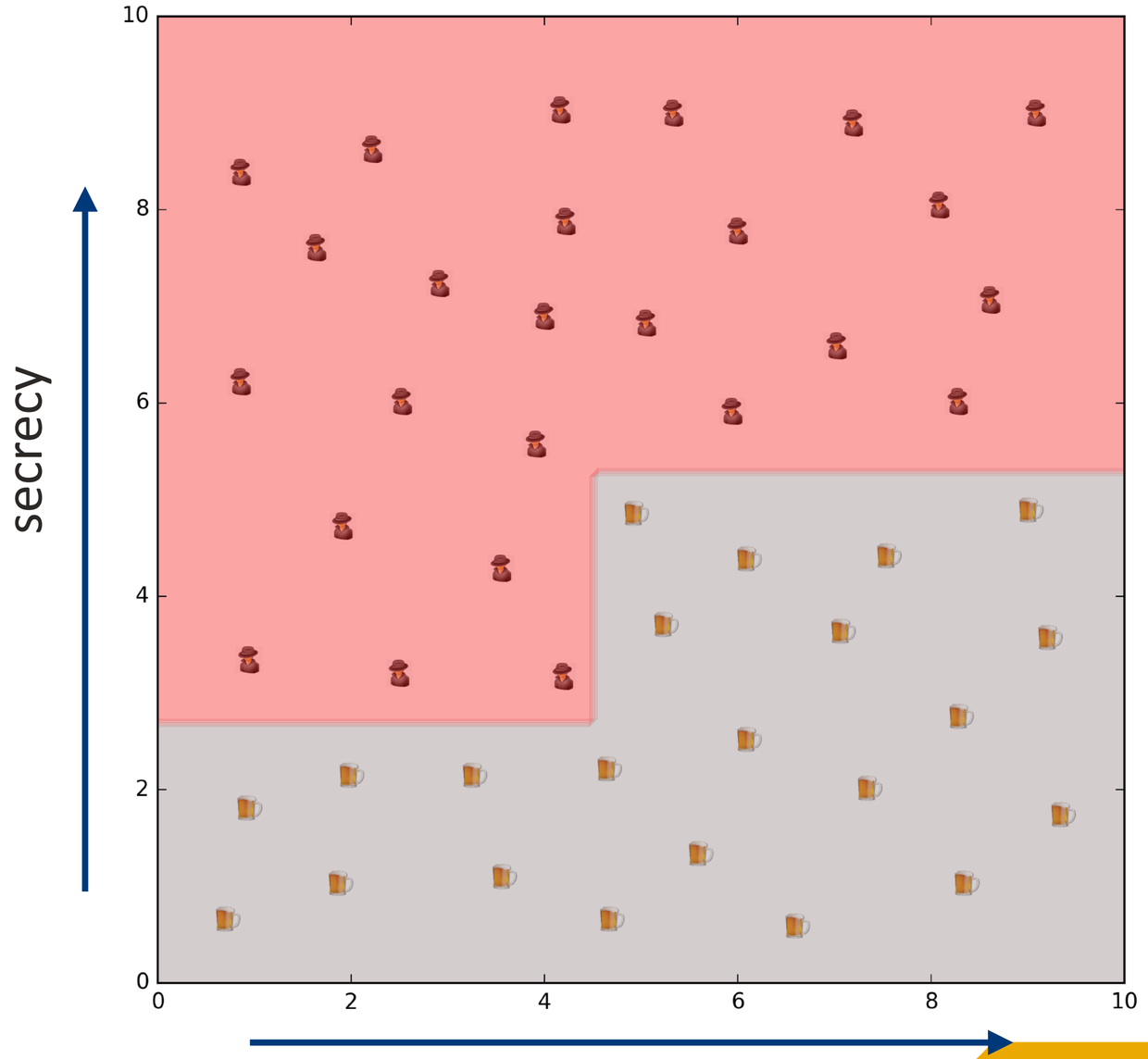


Pen and paper worksheet: solve!



Please take 10 minutes and complete **printed Worksheet - create a Decision Tree by hand!**

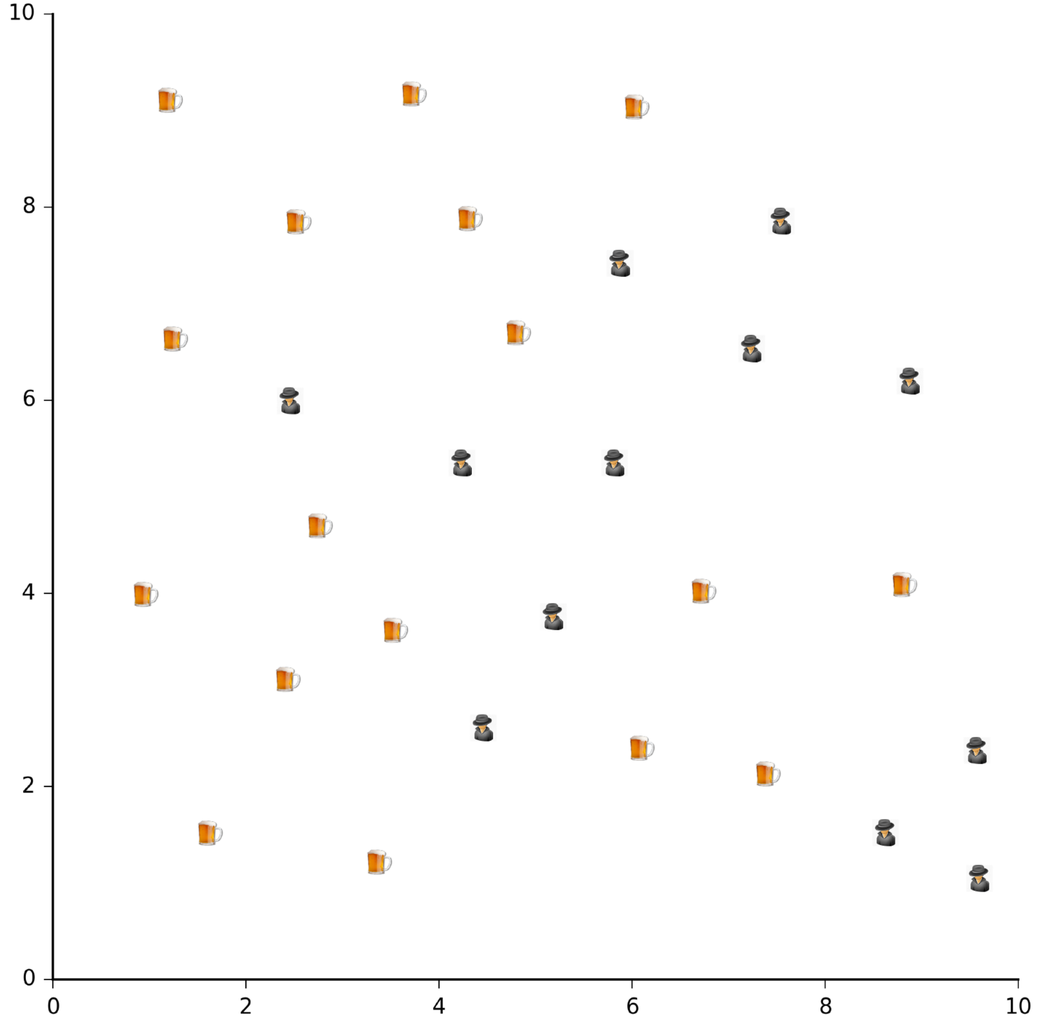
Pen and paper worksheet: solve!



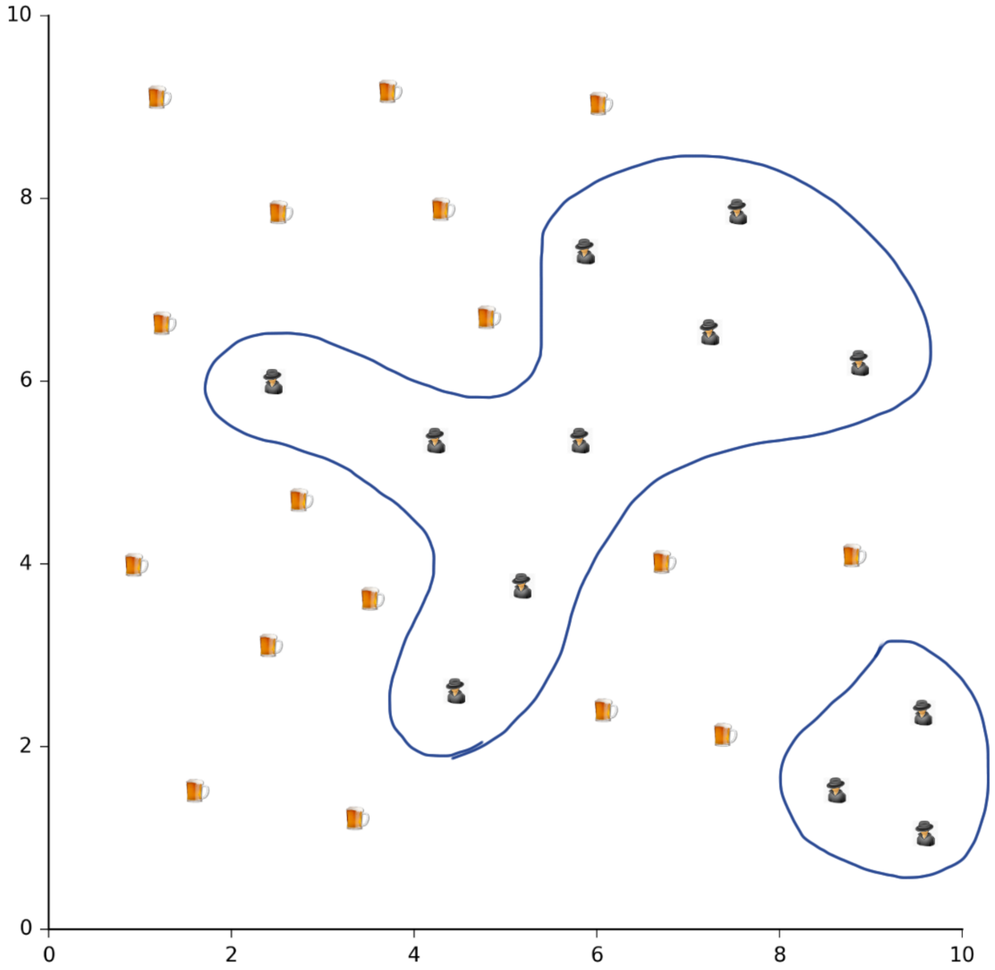
Answer:

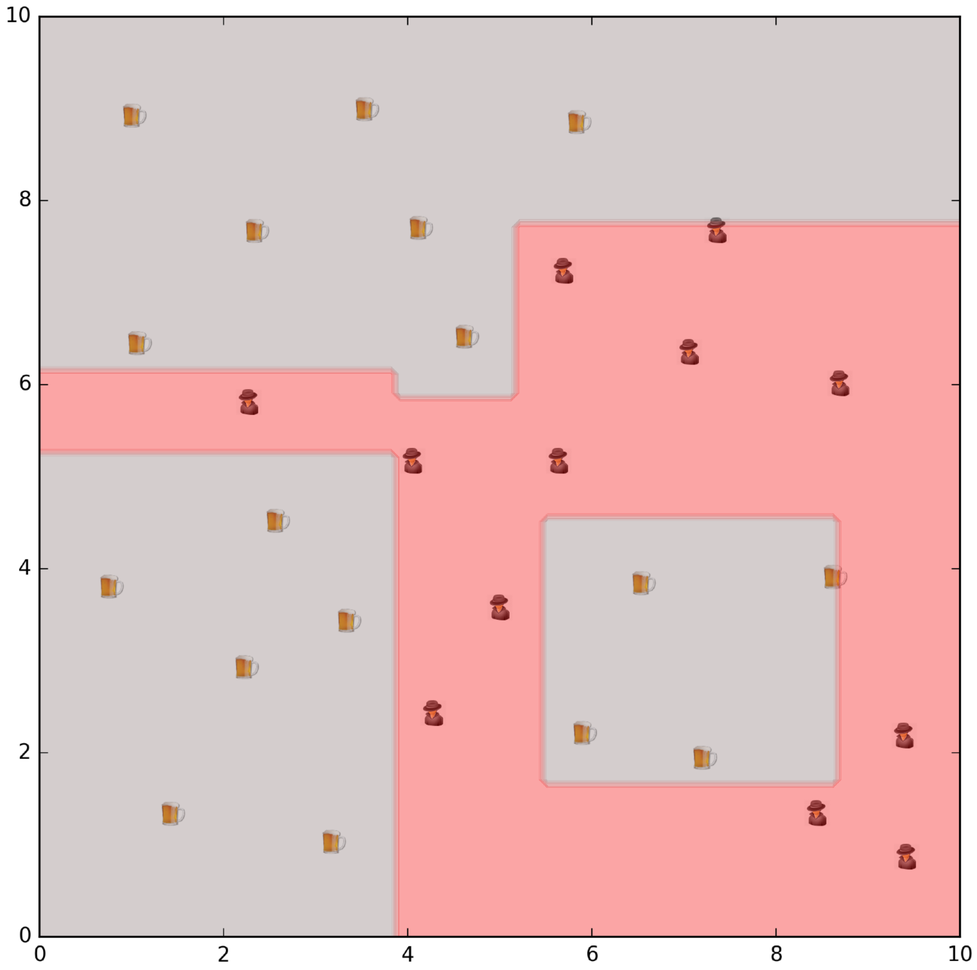
1. secrecy threshold 5.27
2. alcohol threshold 4.47
3. secrecy threshold 2.70

Ultimate last challenge! Who can solve it?

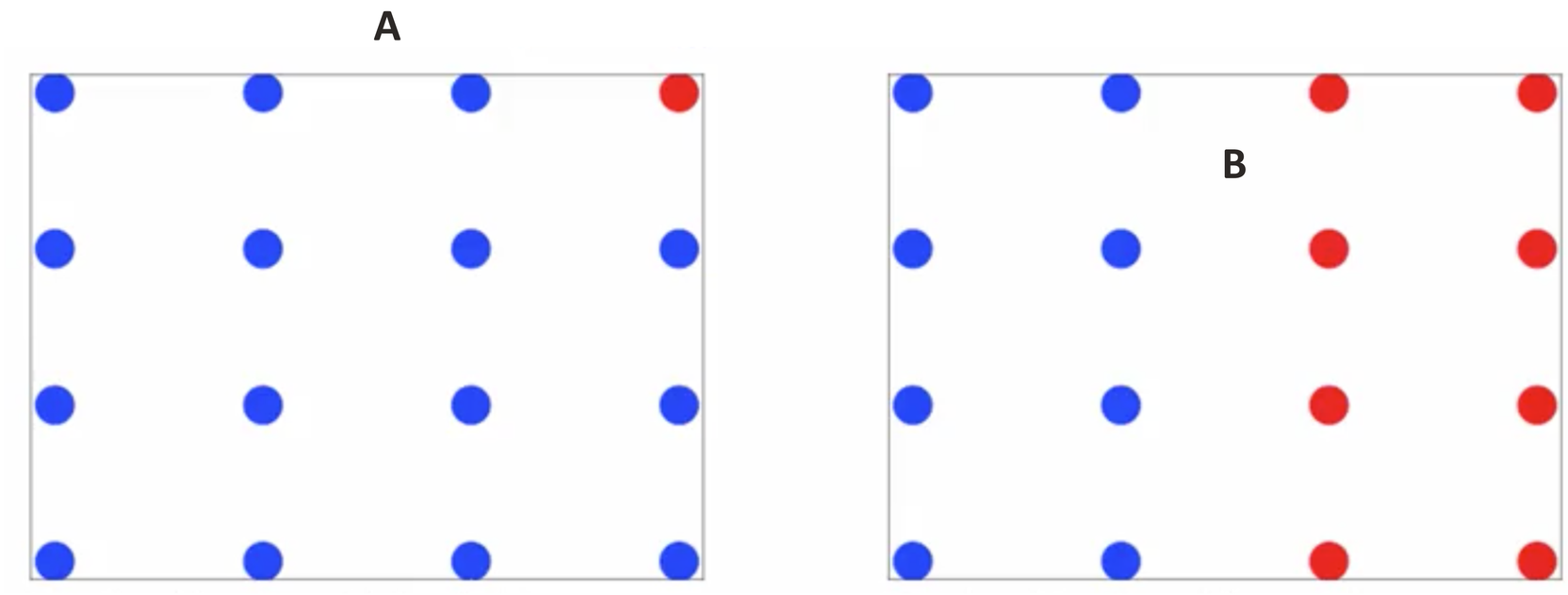


Such boundaries can be a sign of over-fitting!





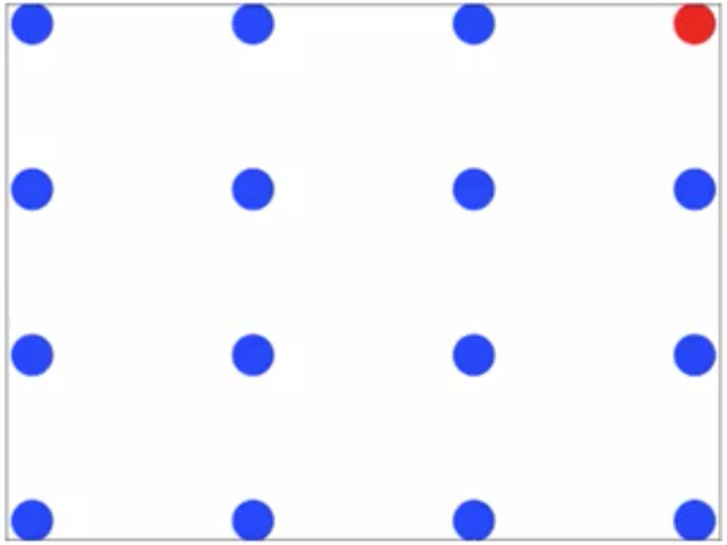
Criterion for best splits: Impurity



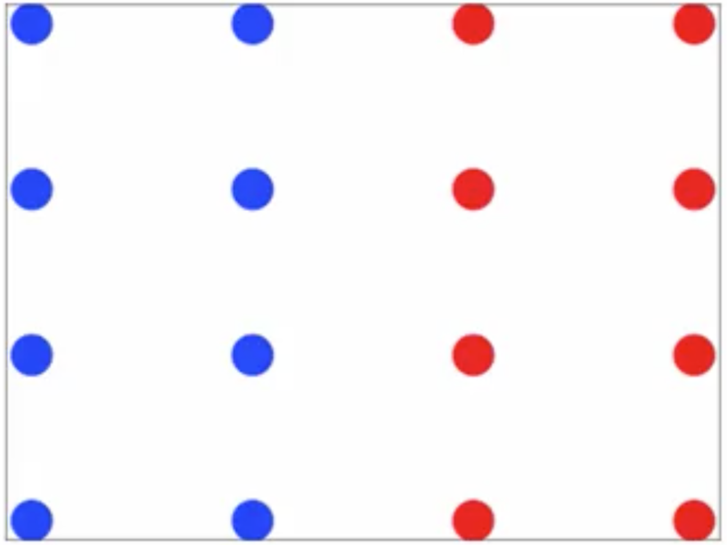
Quiz: By intuition which example appears to be more pure, that is, separates the two classes better?

Criterion for best splits: Information Gain

A



B



Answer: A

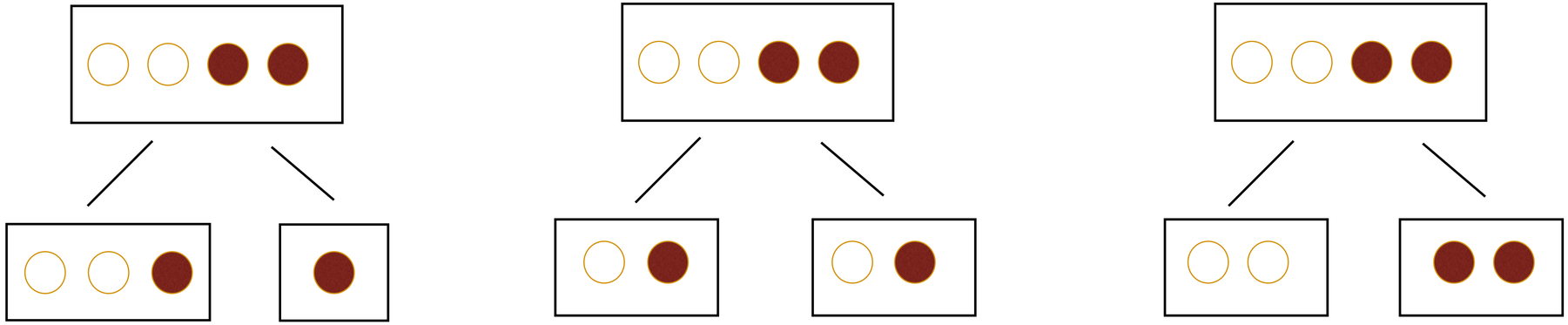
Which feature below gives the best split?

Information Gain = entropy(parent) - [weighted average]entropy(children)

entropy(parent)=1.0

most impure binary system

choosing next best feature to split next



feature 1 feature 2 feature 3

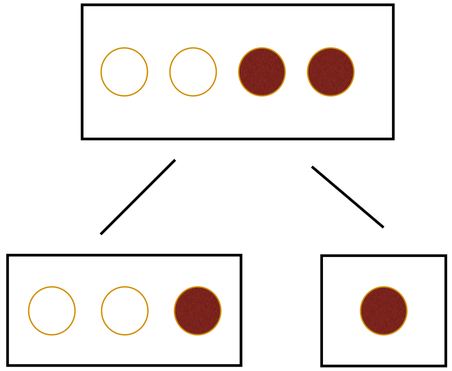
?

Which feature below gives the best split?

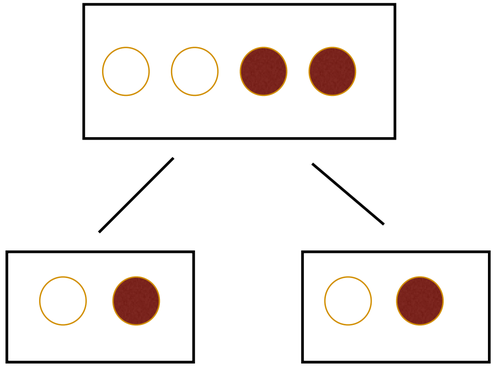
Information Gain = entropy(parent) - [weighted average]entropy(children)

entropy(parent)=1.0
most impure binary system

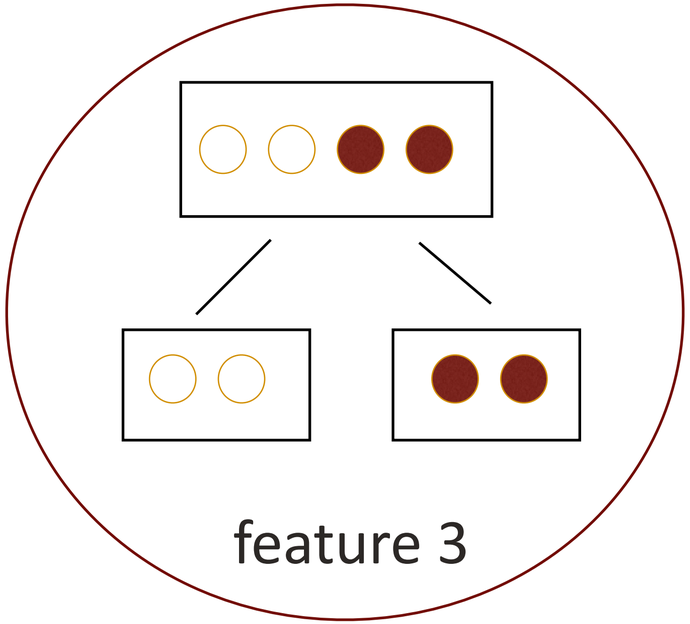
choosing next best feature to split next



feature 1



feature 2



feature 3

Information Gain	**0.3112**	**0**	**1**

Advantages of Decision Trees

- Can be used for regression or classification
- Can be displayed graphically
- Highly interpretable
- Can be specified as a series of rules, and more closely approximate human decision-making than other models
- Prediction is fast
- Features don't need scaling
- Automatically learns feature interactions (they are non-linear models)
- Tend to ignore irrelevant features (especially when there are lots of features)
- Because decision trees are non-linear models they will outperform linear models if the relationship between features and response is highly non-linear

Disadvantages of Decision Trees

- Performance is (generally) not competitive with the best supervised learning methods
- Can easily overfit the training data (tuning is required)
- Small variations in the data can result in a completely different tree (they are high variance models)
- Recursive binary splitting makes "locally optimal" decisions that may not result in a globally optimal tree
- Don't tend to work well if the classes are highly unbalanced
- Don't tend to work well with very small datasets

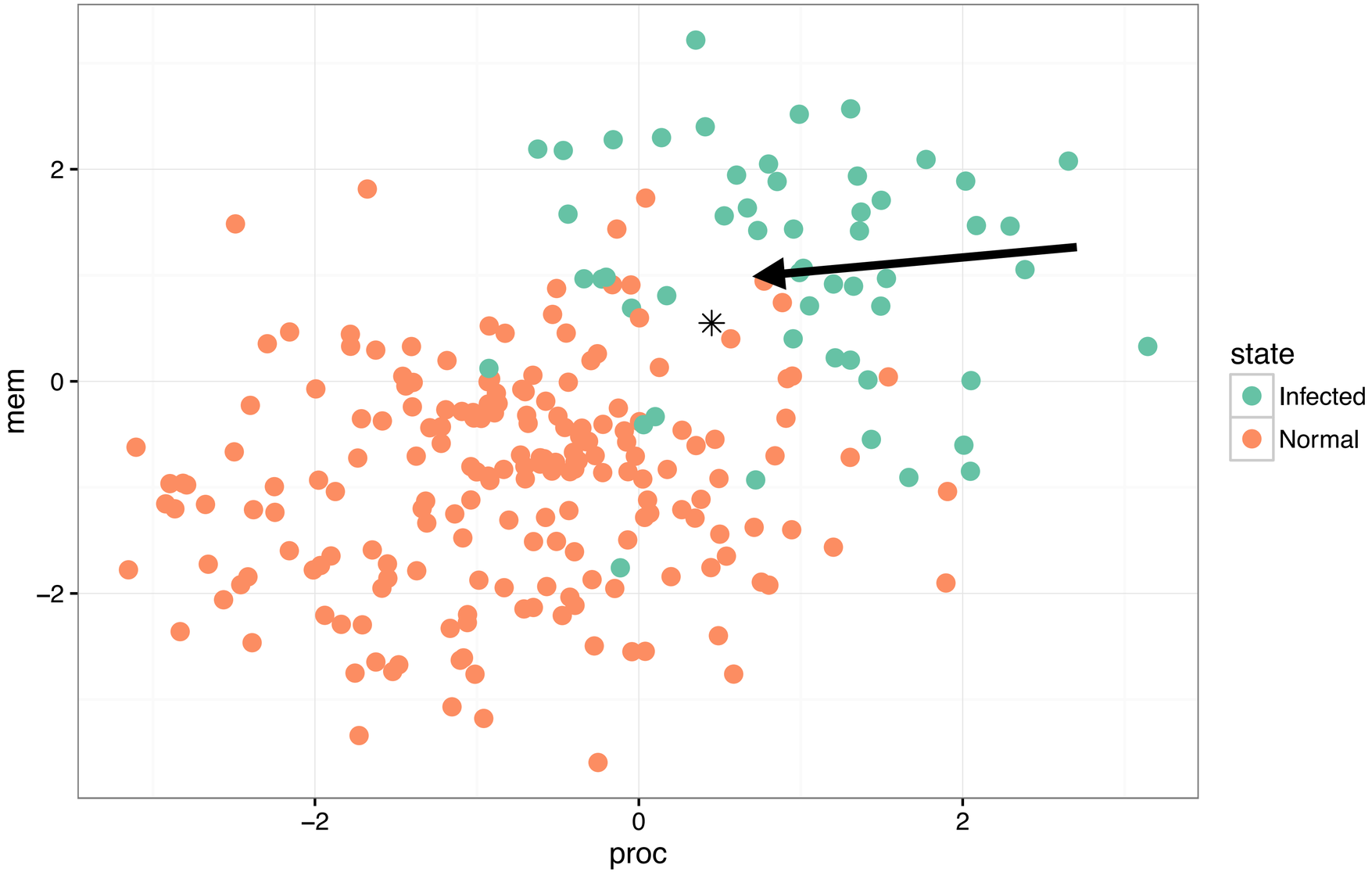
K-Nearest Neighbors Classifier

K-nearest neighbors

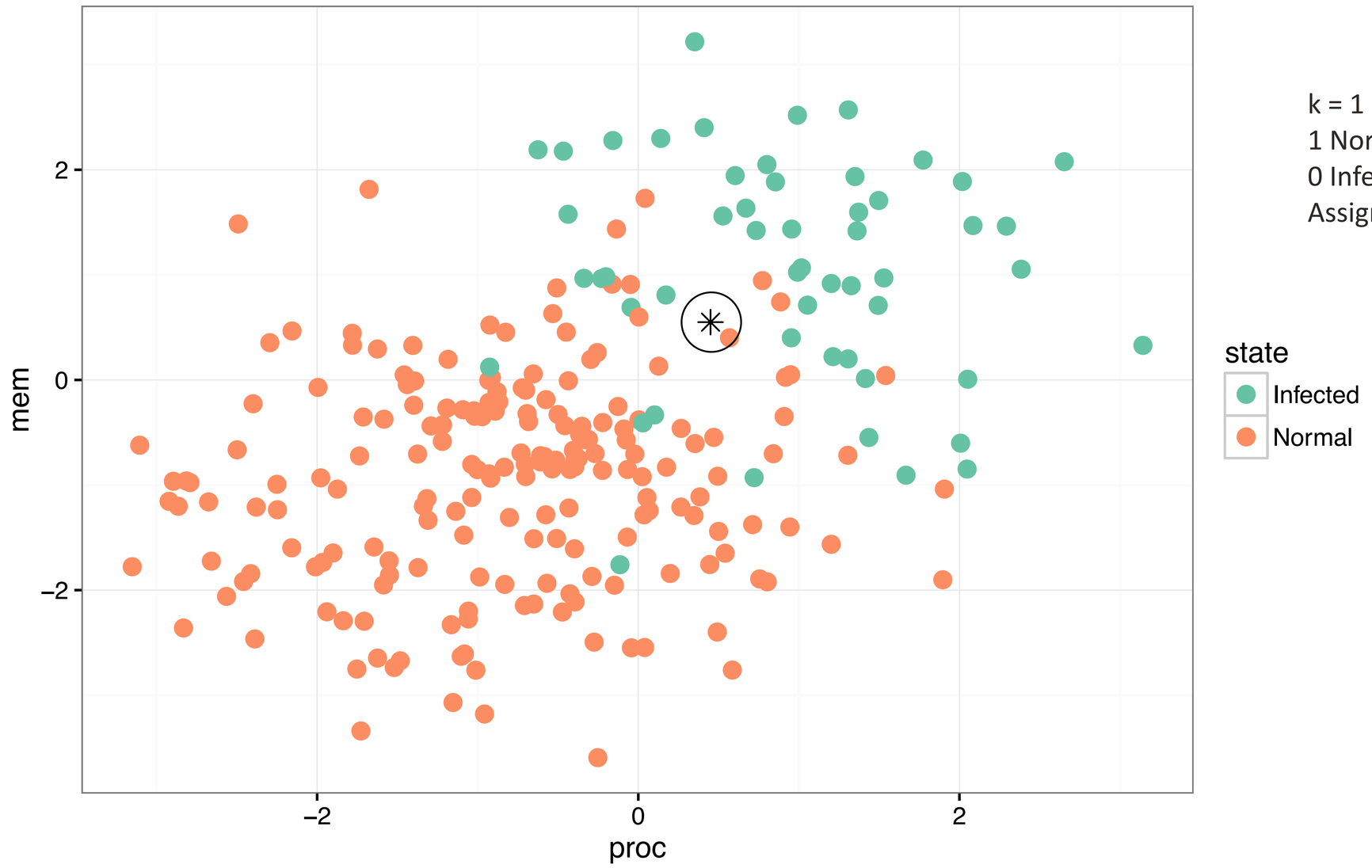
Define K, and for every unknown observation:

1. Find k-nearest neighbors (by *distance*)
2. Assign class based on dominate class
3. Optional weight by distance

K-nearest neighbors

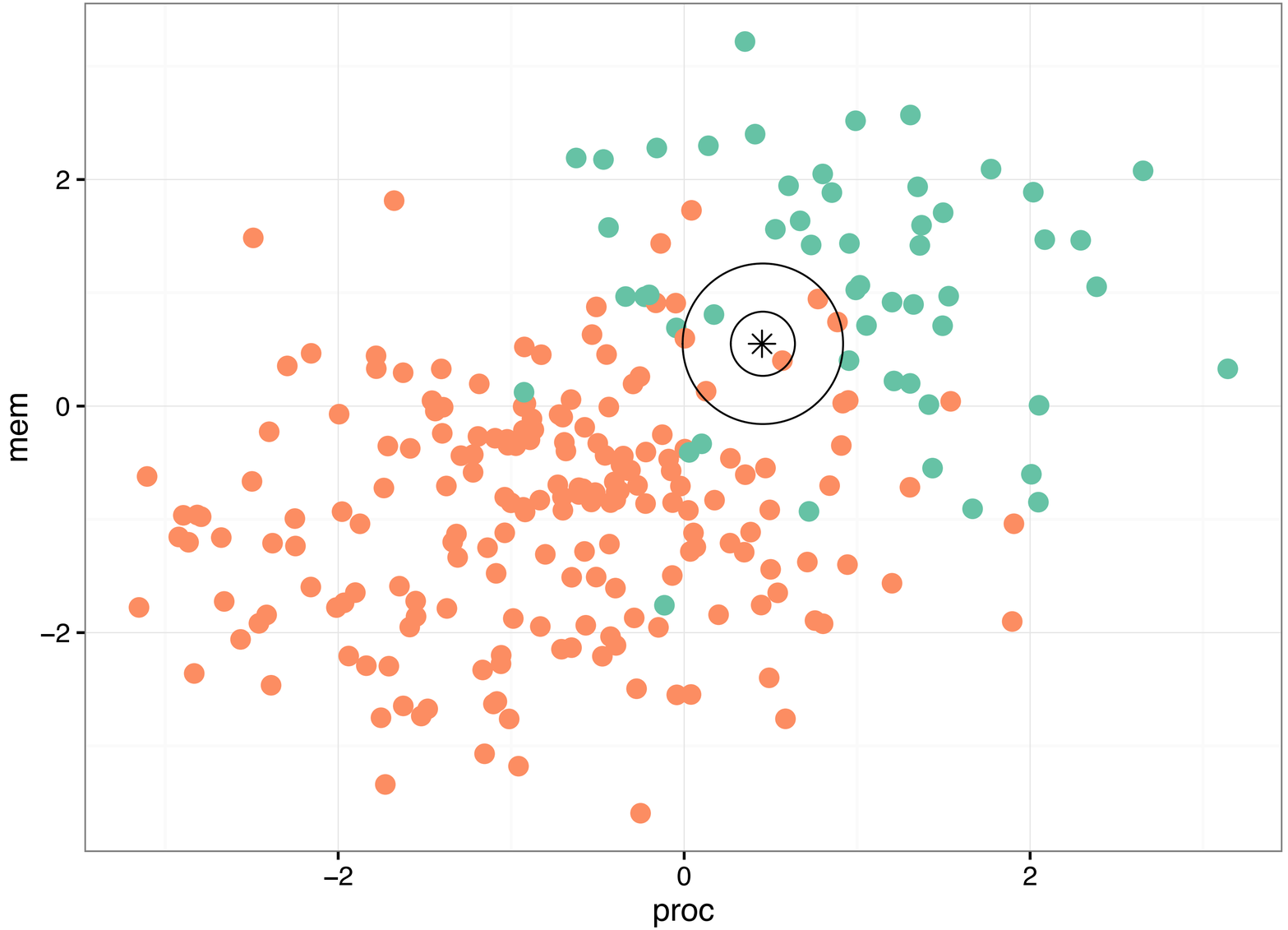


K-nearest neighbors



k = 1
1 Normal
0 Infected
Assign “Normal”

K-nearest neighbors



k = 5
4 Normal
1 Infected
Assign “Normal”

state
Infected
Normal

K-nearest neighbors

Further reading: https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/

K-NN Classifiers

Advantages of K-NN Classifiers

- Robust to noisy, non-linear training data
- Flexible to feature / distance choices
- Works with large datasets
- Naturally handles multi-class cases

Disadvantages of K-NN Classifiers:

- Need to determine the value of K
- Training is fast but predictions are slow. Indexing can help
- Must have meaningful distance function

Ensembles

Ensembles

Ensemble learning is the process of combining several predictive models in order to produce a combined model that is more accurate than any individual model.

- **Regression:** take the average of the predictions
- **Classification:** take a vote and use the most common prediction, or take the average of the predicted probabilities

Ensembles

For ensembling to work well, the models must have the following characteristics:

- **Accurate:** they outperform random guessing
- **Independent:** their predictions are generated using different processes

The big idea: If you have a collection of individually imperfect (and independent) models, the "one-off" mistakes made by each model are probably not going to be made by the rest of the models, and thus the mistakes will be discarded when averaging the models.

Note: As you add more models to the voting process, the probability of error decreases, which is known as Condorcet's Jury Theorem.

Bagging

1. Grow *n* trees using *n* bootstrap samples from the training data.
2. Train each tree on its bootstrap sample and make predictions.
3. Combine the predictions:
   - Average the predictions for **regression trees**
   - Take a majority vote for **classification trees**

Random Forest

Random Forests are a **slight variation of bagged trees** that have even better performance:

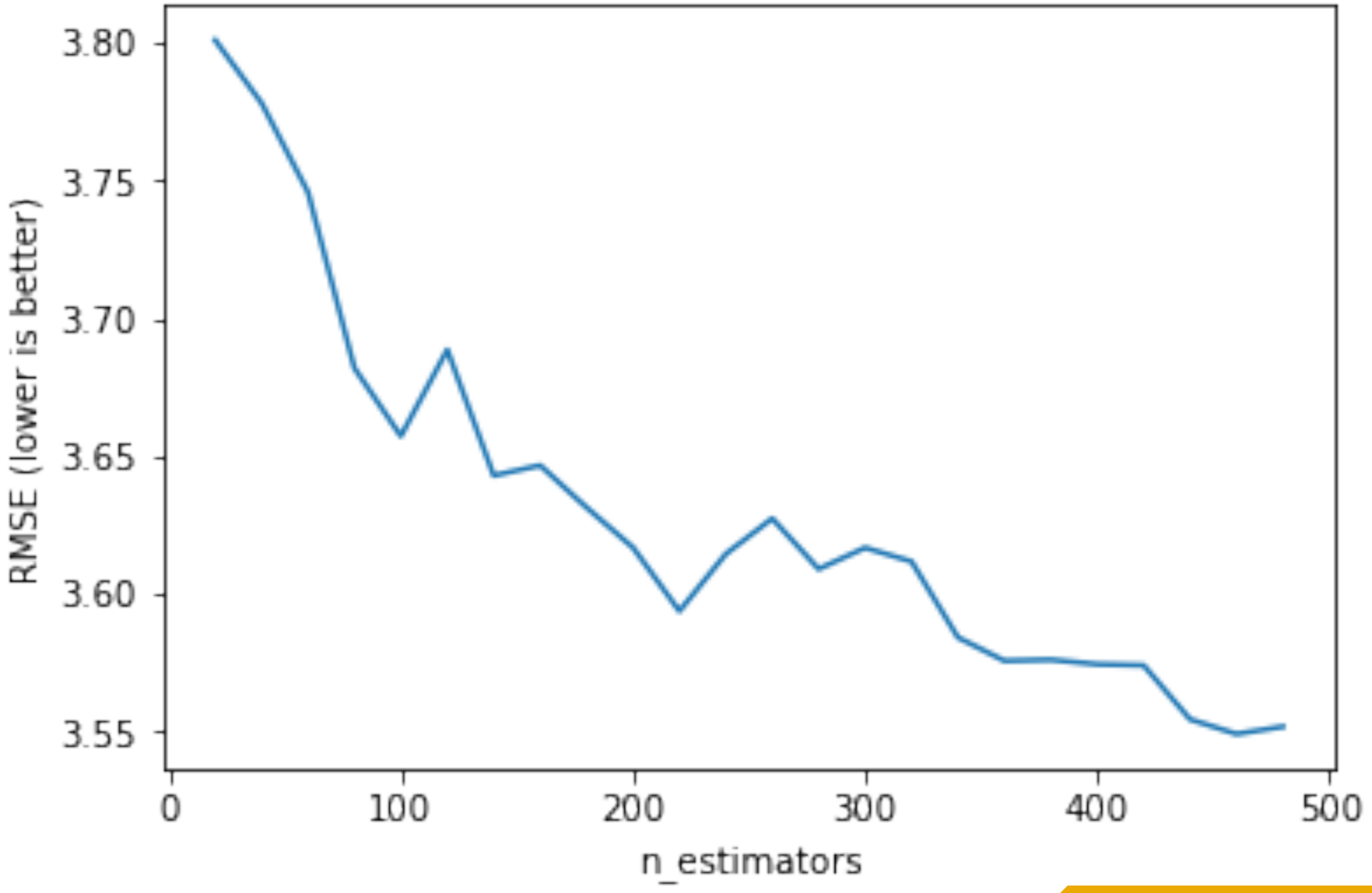
- Just like bagging, we create an ensemble of decision trees using bootstrapped samples of the training set.
- However, when building each tree, each time a split is considered, a **random sample of *m* features** is chosen as split candidates from the **full set of *p* features**. The split is only allowed to use **one of those *m* features**.
    - A new random sample of features is chosen for **every single tree at every single split**.
    - For **classification**, m is typically chosen to be the square root of p (the total number of features).
    - For **regression**, m is typically chosen to be somewhere between $p/3$ and p.

Tuning a Random Forest

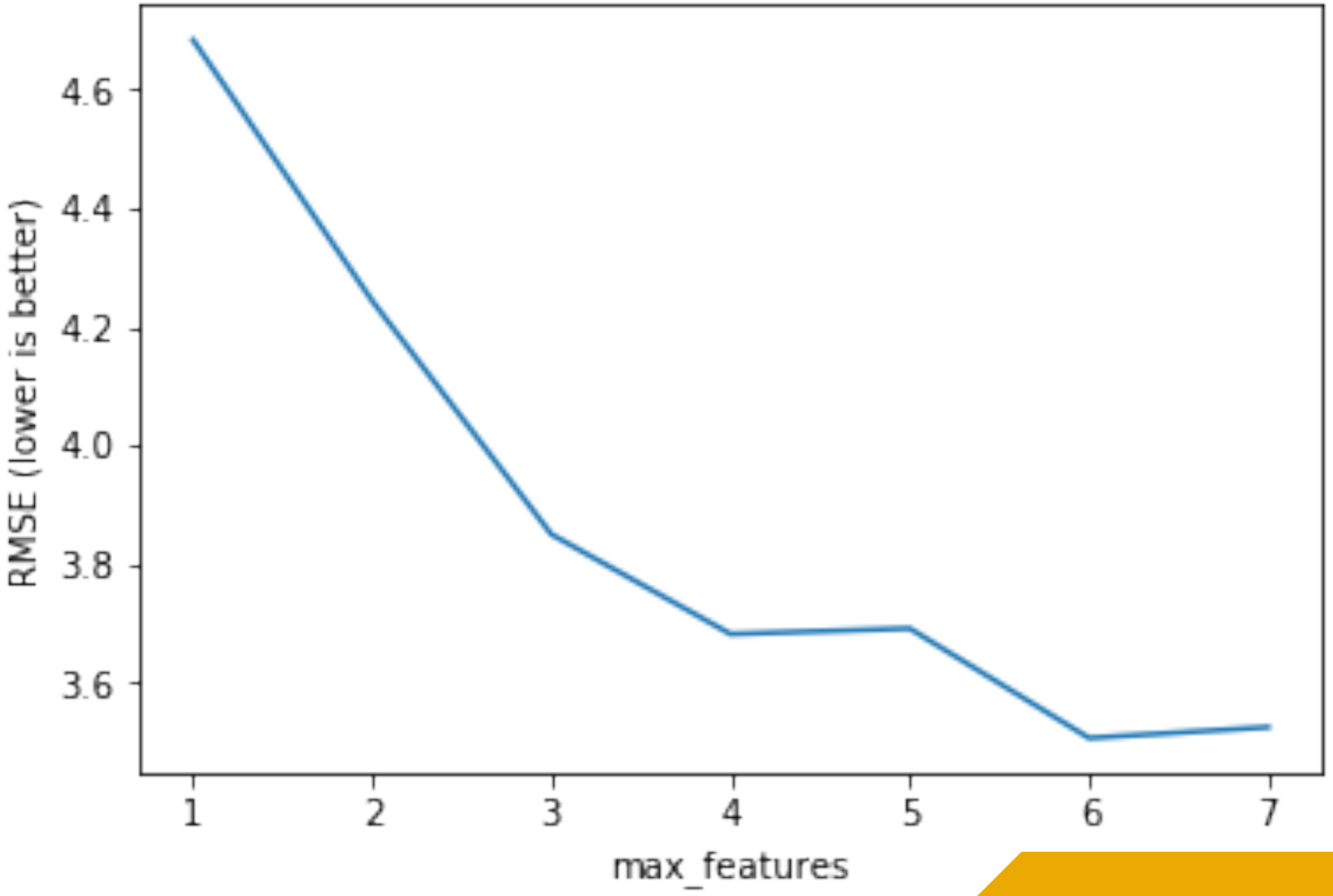
2 important parameters that should be tuned when creating a random forest model are:

- The number of trees to grow (called **n_estimators** in scikit-learn)
- The number of features that should be considered at each split (called **max_features** in scikit-learn)

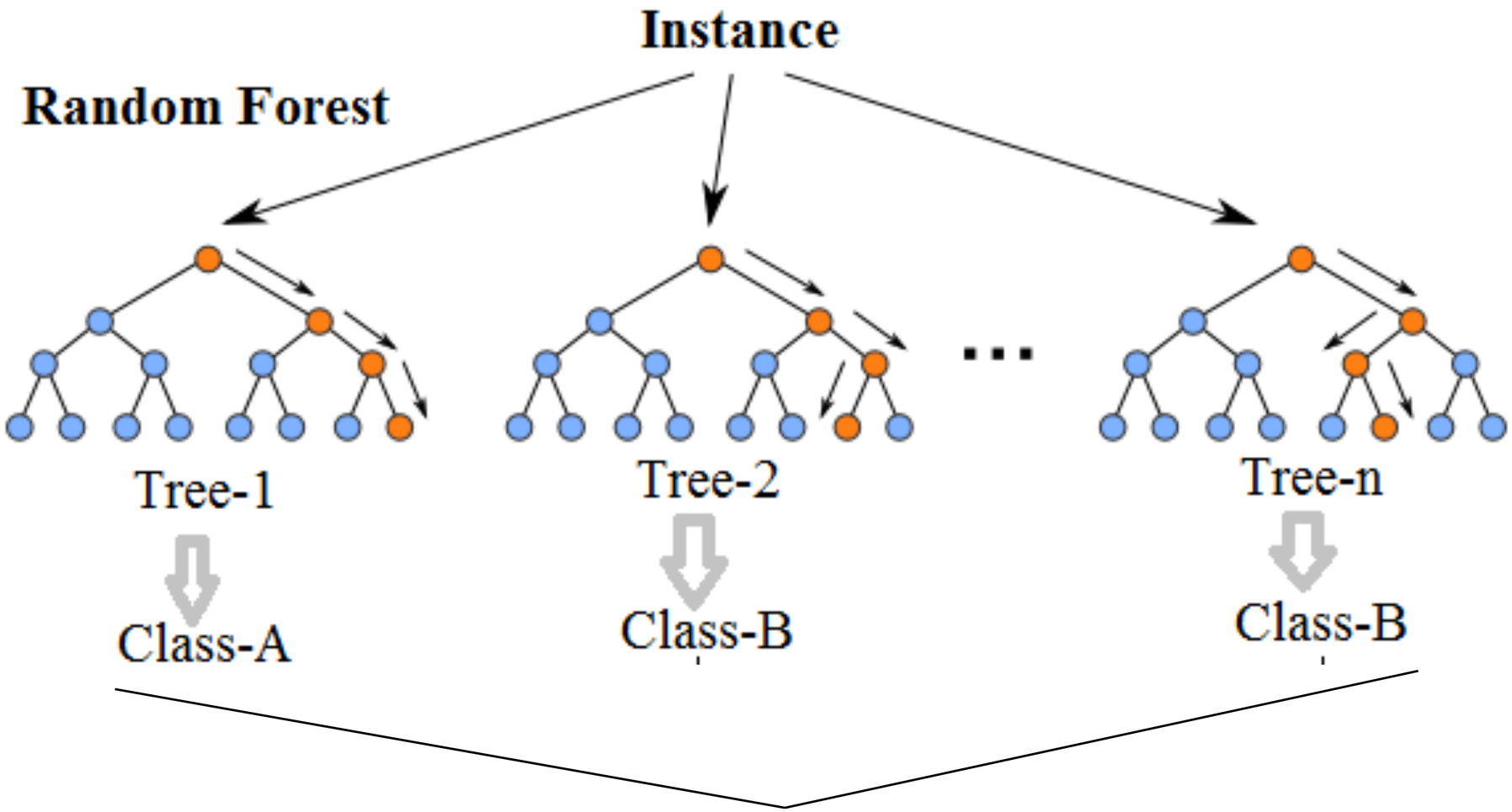
Tuning a Random Forest



Tuning a Random Forest



Tuning a Random Forest



Class A: p% Class B: 1-p%

Majority Win

Random Forests

Advantages of Random Forests:

- Performance is competitive with the best supervised learning methods
- Provides a more reliable estimate of feature importance
- Allows you to estimate out-of-sample error without using train/test split or cross-validation

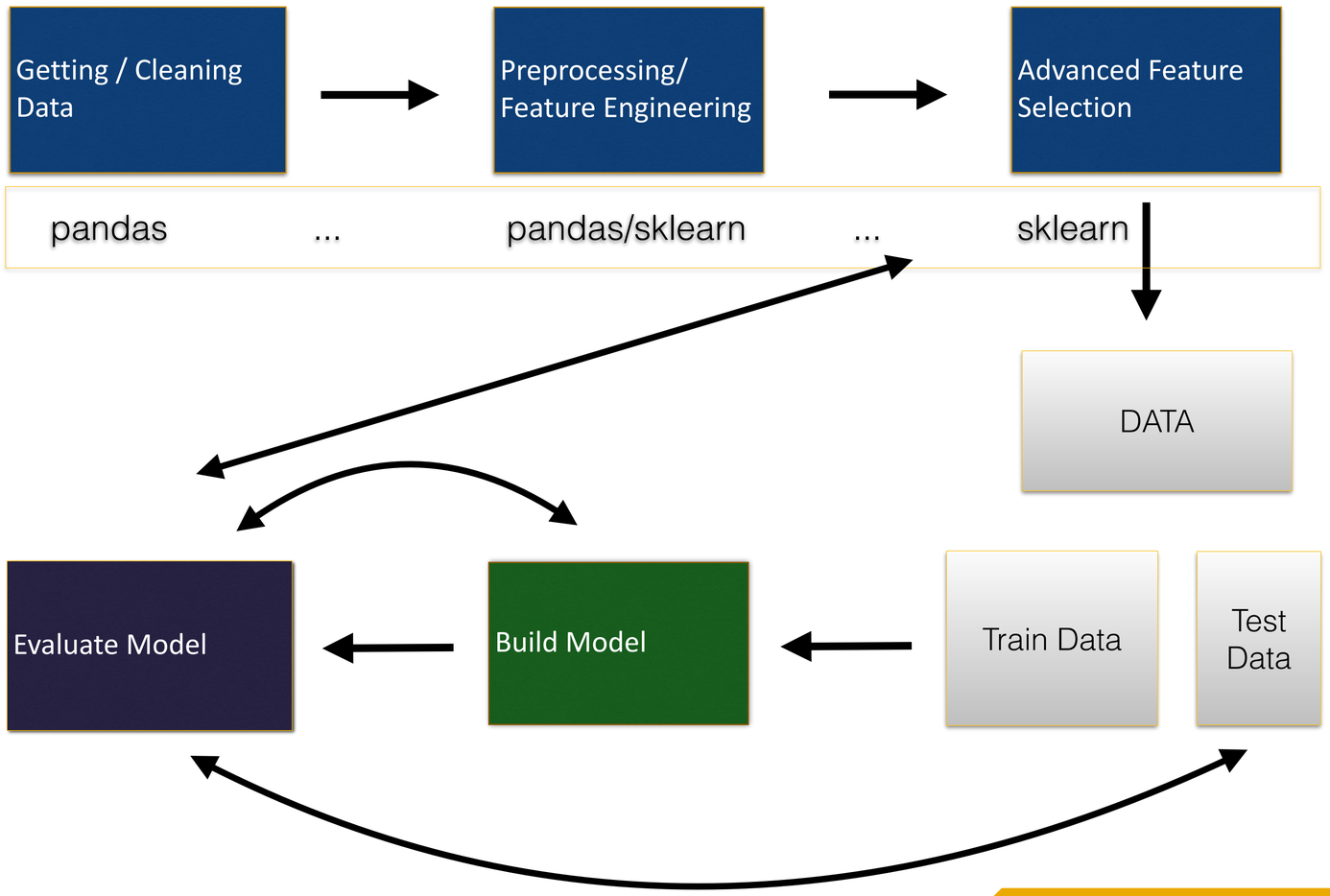
Disadvantages of Random Forests:

- Less interpretable
- Slower to train
- Slower to predict

Supervised Machine Learning

Implementation

Machine Learning Process



Scikit-Learn Process

The basic process for any supervised learning is the same for any supervised learning in Python.

1. Create the Classifier or Regressor object.

```
clf = RandomForestClassifier()
```

2. Call the `.fit()` method using the **training feature matrix (X)** and the **training target vector (y)**.

```
clf.fit( training_features, training_target )
```

3. Call the `.predict()` method using a feature matrix

#Returns vector of predictions

```
clf.predict( testing_features )
```

Train-Predict in sklearn

```
## Support Vector Machine Classification

clf = svm.SVC()
clf = clf.fit(features, target)
predictions = clf.predict(features)
```

clf means
classifier

Train-Predict in sklearn

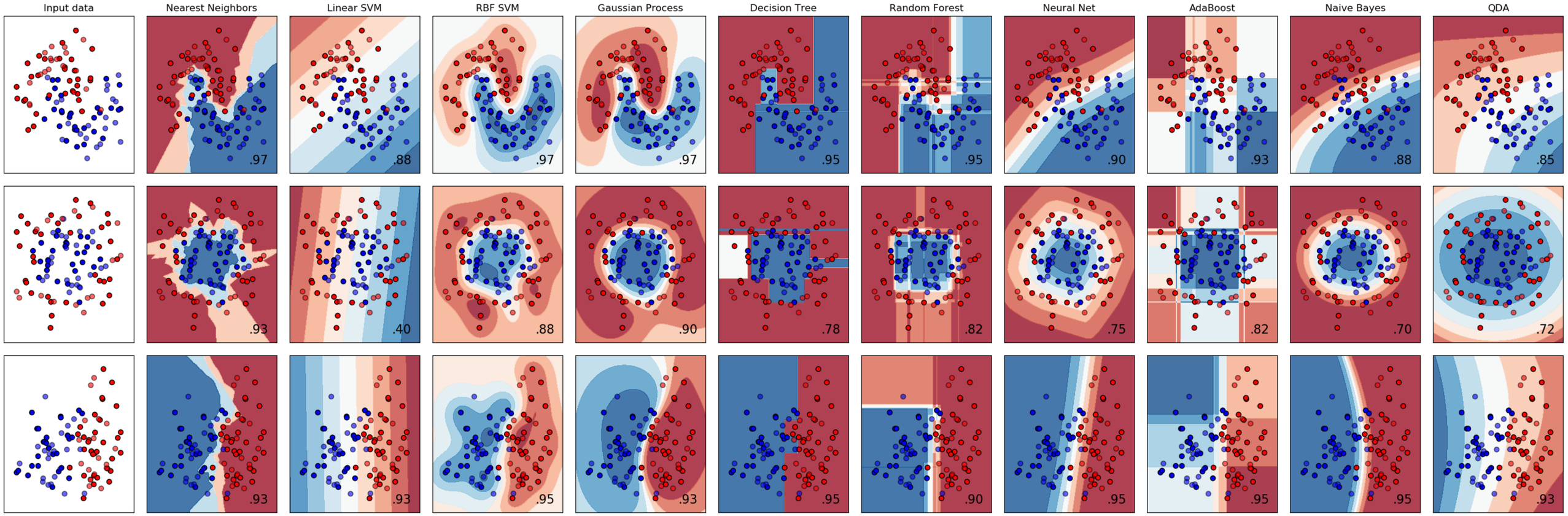
```
## Decision Tree Classification

clf = tree.DecisionTreeClassifier()
clf = clf.fit(features, target)
predictions = clf.predict(features)
```

clf means
classifier

Classification Algorithms in Scikit-Learn

- Generalized Linear Models
- Kernel Ridge
- Support Vector Machines
- Nearest Neighbors
- Gaussian Processes
- Naive Bayes
- Trees
- Neural Networks
- AdaBoost
- Gradient Tree Boosting
- Ensemble methods

Classification Algorithms in Scikit-Learn



http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html

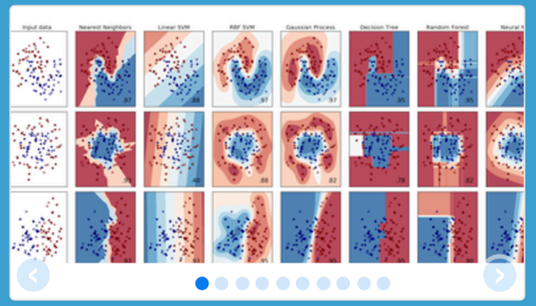
http://scikit-learn.org/stable/

scikit learn

Home Installation Documentation Examples

powered by Google

Google Custom Search Search



scikit-learn

Machine Learning in Python

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Classification

Identifying to which category an object belongs to.

Applications: Spam detection, Image recognition.
Algorithms: SVM, nearest neighbors, random forest, ...

— Examples

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.
Algorithms: SVR, ridge regression, Lasso, ...

— Examples

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, ...

— Examples

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency
Algorithms: PCA, feature selection, non-negative matrix factorization.

— Examples

Model selection

Comparing, validating and choosing parameters and models.

Goal: Improved accuracy via parameter tuning
Modules: grid search, cross validation, metrics.

— Examples

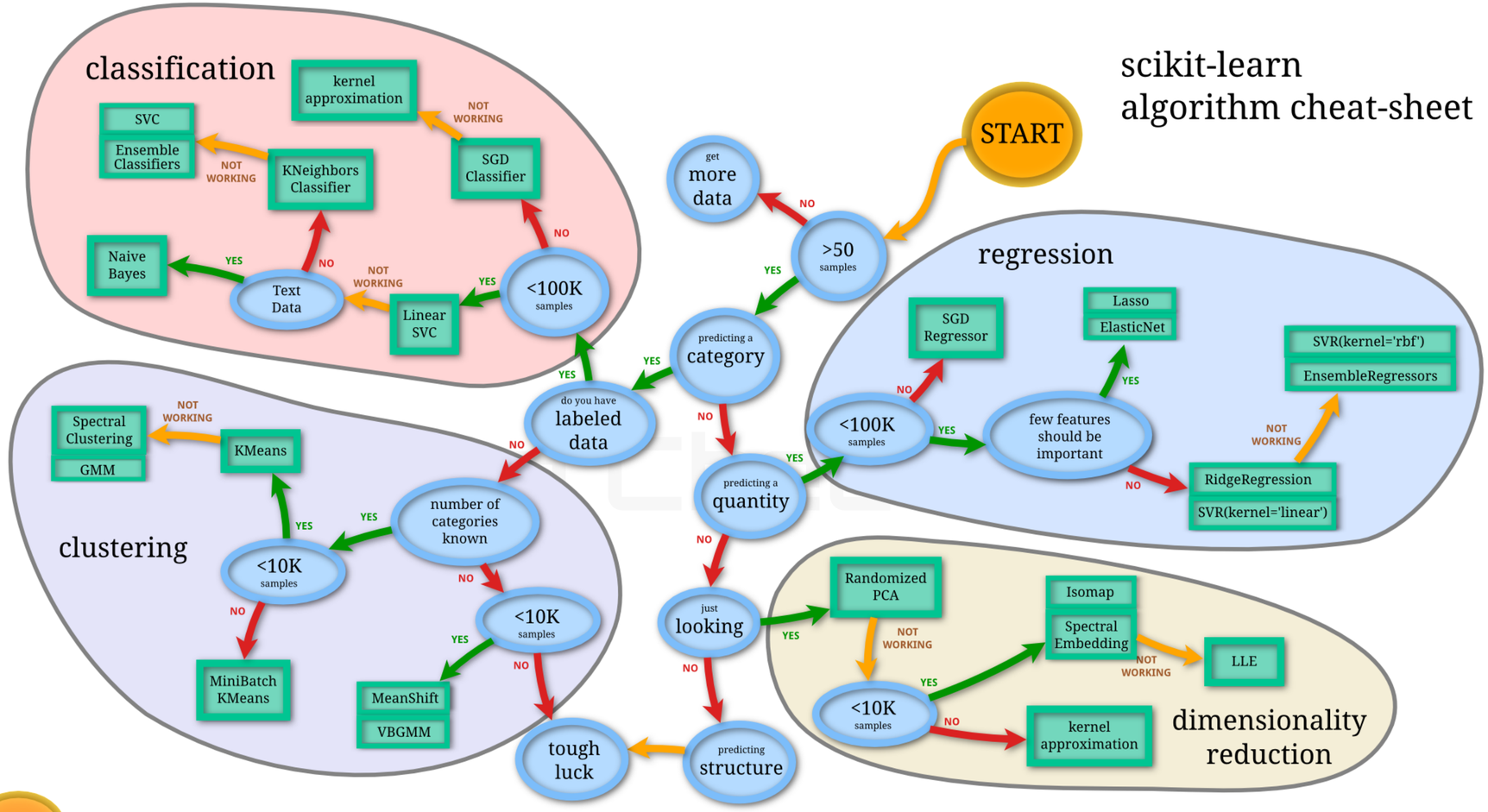
Preprocessing

Feature extraction and normalization.

Application: Transforming input data such as text for use with machine learning algorithms.
Modules: preprocessing, feature extraction.

— Examples

scikit-learn algorithm cheat-sheet



Back

Sklearn Classifiers Navigation

sklearn.tree.DecisionTreeClassifier

class sklearn.tree. **DecisionTreeClassifier** (*criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_split=1e-07, class_weight=None, presort=False*) [source]

A decision tree classifier.

Read more in the User Guide.

Parameters:

criterion : string, optional (default="gini")

The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

splitter : string, optional (default="best")

The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_features : int, float, string or None, optional (default=None)

The number of features to consider when looking for the best split:

Check parameters for each classifier!

Sklearn Classifiers Navigation

fit (*X, y, sample_weight=None, check_input=True, X_idx_sorted=None*) [source]

Build a decision tree classifier from the training set (X, y).

Parameters: **X** : array-like or sparse matrix, shape = [n_samples, n_features]

The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y : array-like, shape = [n_samples] or [n_samples, n_outputs]

The target values (class labels) as integers or strings.

sample_weight : array-like, shape = [n_samples] or None

Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

predict (*X, check_input=True*) [source]

Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

Parameters: **X** : array-like or sparse matrix of shape = [n_samples, n_features]

The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input : boolean, (default=True)

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns: **y** : array of shape = [n_samples] or [n_samples, n_outputs]

The predicted classes, or the predict values.

Check input and output
dimensions of fit and predict methods for feature matrix X and target vector y!

Sklearn Classifiers Navigation

Attributes:

classes_ : array of shape = [n_classes] or a list of such arrays

The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

feature_importances_ : array of shape = [n_features]

The feature importances. The higher, the more important the feature. The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance [R245].

max_features_ : int,

The inferred value of max_features.

n_classes_ : int or list

The number of classes (for single output problems), or a list containing the number of classes for each output (for multi-output problems).

n_features_ : int

The number of features when `fit` is performed.

After calling fit method you can retrieve certain attributes of your clf object!

Sklearn Classifiers Navigation

fit (*X, y=None*) [source]

Compute k-means clustering.

Parameters:	**X** : array-like or sparse matrix, shape=(n_samples, n_features) Training instances to cluster.

fit_predict (*X, y=None*) [source]

Compute cluster centers and predict cluster index for each sample.

Convenience method; equivalent to calling fit(X) followed by predict(X).

fit_transform (*X, y=None*) [source]

Compute clustering and transform X to cluster-distance space.

Equivalent to fit(X).transform(X), but more efficiently implemented.

transform (*X, y=None*) [source]

Transform X to a cluster-distance space.

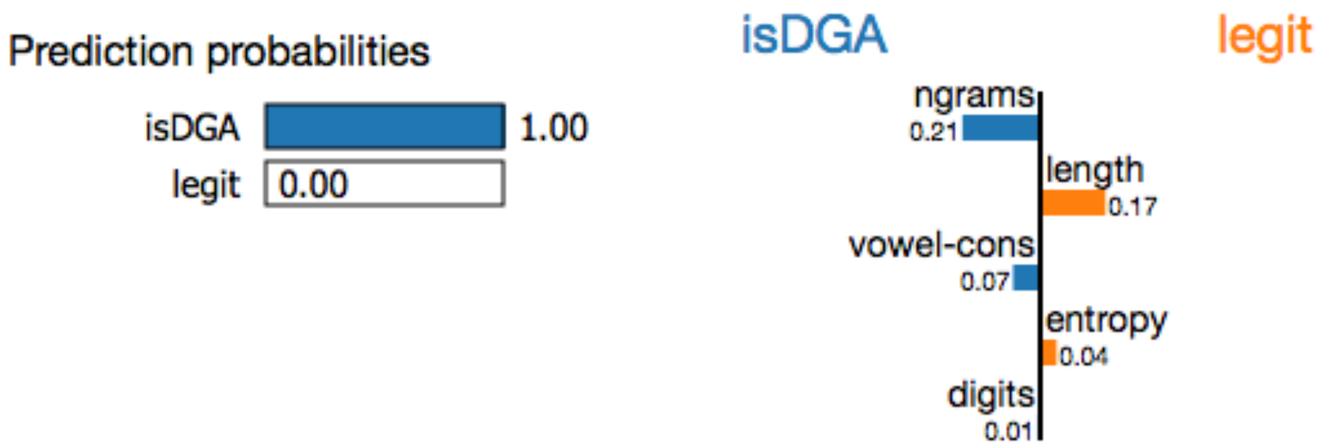
In the new space, each dimension is the distance to the cluster centers. Note that even if X is sparse, the array returned by transform will typically be dense.

Parameters:	**X** : {array-like, sparse matrix}, shape = [n_samples, n_features] New data to transform.
Returns:	**X_new** : array, shape [n_samples, k] X transformed in the new space.

Clustering and dimensionality reduction methods like PCA have more fit and transform rather than predict and often combine fit_transform in one method!

"Why should I trust you?"
Explaining the predictions of any classifier

```
import lime
import lime.lime_tabular
explainer = lime.lime_tabular.LimeTabularExplainer(<training_data>,
        feature_names=<feature names>,
        class_names=<class names>,
        discretize_continuous=False)

exp = explainer.explain_instance( <test_row>,
   <model>.predict_proba,
   num_features=5,
   top_labels=1)

exp.show_in_notebook(show_table=True, show_all=False)
```



Feature	Value
length	6.00
digits	0.00
entropy	2.58
vowel-cons	0.50
ngrams	1606.84

https://github.com/marcotcr/lime

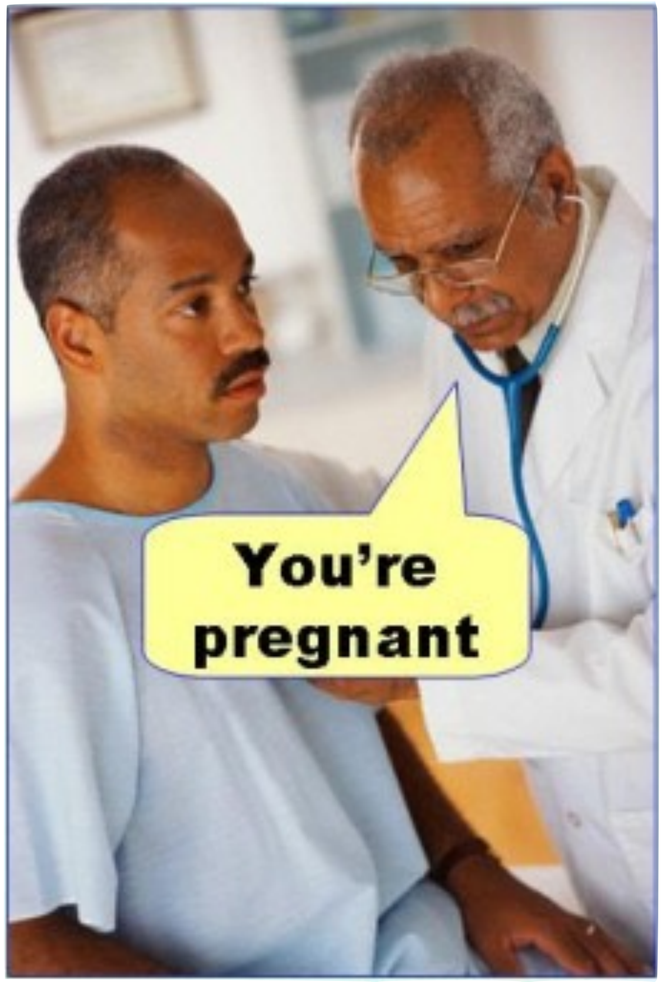
Supervised Machine Learning

Metrics and cross-validation

Performance Metrics for Classifiers

Type I Error
(False Positive)

Type II Error
(False Negative)



Confusion Matrix

```
target =      [1,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0]
target_pred = [0,1,1,0,0,1,1,1,1,0,0,0,0,0,0,0]
print(confusion_matrix(target, target_pred))
```

True target

0 [[7 2]
1 [3 4]]

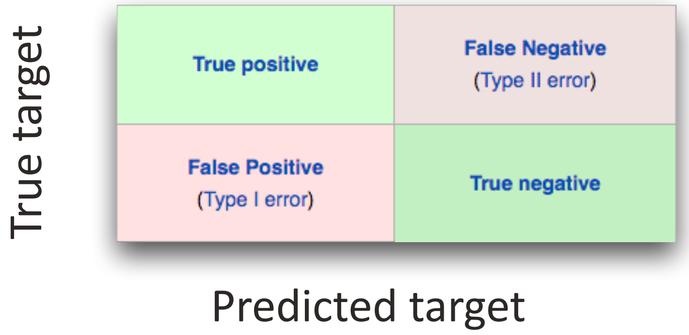
0 1

Predicted target

True positive	**False Negative** (Type II error)
False Positive (Type I error)	**True negative**

Accuracy

```
accuracy = accuracy_score(target, target_pred)
```



$$\begin{array}{cc} & \begin{array}{cc} 0 & 1 \end{array} \\ \begin{array}{c} 0 \\ 1 \end{array} & \begin{bmatrix} 7 & 2 \\ 3 & 4 \end{bmatrix} \end{array}$$

Accuracy = (TP + TN)/ N

Quiz: Calculate by hand!

Accuracy

```
accuracy = accuracy_score(target, target_pred)
```

True target

True positive	**False Negative** (Type II error)
False Positive (Type I error)	**True negative**

Predicted target

$$\begin{array}{c} 0 \\ 1 \end{array} \begin{bmatrix} 7 & 2 \\ 3 & 4 \end{bmatrix}$$

0 1

Accuracy = (TP + TN)/ N

= (7+4)/16

= 0.6875

Precision

Column-wise!

Precision

```
precision_class = precision_score(target, target_pred, average = None)
precision_avg = precision_score(target, target_pred, average = 'binary')
```

True target

0 [[7 2]
1 [3 4]]

0 1

Predicted target

True positive	**False Negative** (Type II error)
False Positive (Type I error)	**True negative**

Precision = TP/ (TP + FP)

Quiz: Calculate precision by hand for both classes!

Precision

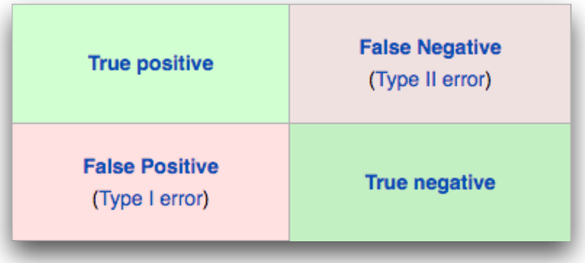
```
precision_class = precision_score(target, target_pred, average = None)
precision_avg = precision_score(target, target_pred, average = 'binary')
```

True target

0 [[7 2]
1 [3 4]]

0 1

Predicted target



Precision = TP/ (TP + FP)

Prec_Class0 = 7/(7+3) = 0.7

Prec_Class1 = 4/(4+2) = 0.666

Accurate
Precise

Not Accurate
Precise

Accurate
Not Precise

Not Accurate
Not Precise

Recall

Row-wise!

Recall

```
recall_class = recall_score(target, target_pred, average = None)
recall_avg = recall_score(target, target_pred, average = 'binary')
```

True target

0 [[7 2]
1 [3 4]]

0 1

Predicted target

True positive	**False Negative** (Type II error)
False Positive (Type I error)	**True negative**

Recall = TP/ (TP + FN)

Quiz: Calculate recall by hand for both classes!

Recall

```
recall_class = recall_score(target, target_pred, average = None)
recall_avg = recall_score(target, target_pred, average = 'binary')
```

True target

	0	1
0	7	2
1	3	4

Predicted target

True positive	**False Negative** (Type II error)
False Positive (Type I error)	**True negative**

Recall = TP/ (TP + FN)

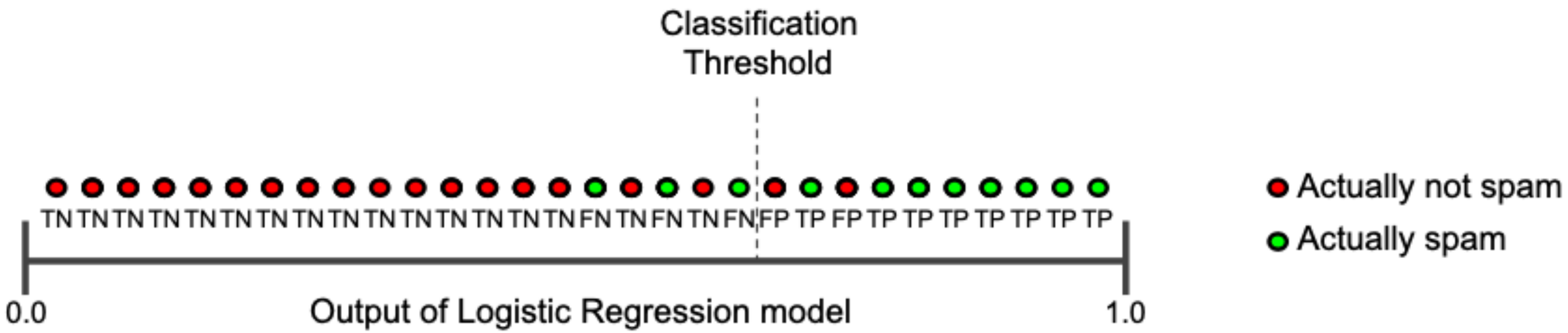
Rec_Class0 = 7/(7+2) = 0.777

Rec_Class1 = 4/(4+3) = 0.571

Precision vs. Recall

- Precision and Recall are often inversely related--as precision increases, often recall goes down.
- Precision is the proportion of positive identifications thet are actually correct
- Recall is the proportion of actual positives was identified correctly

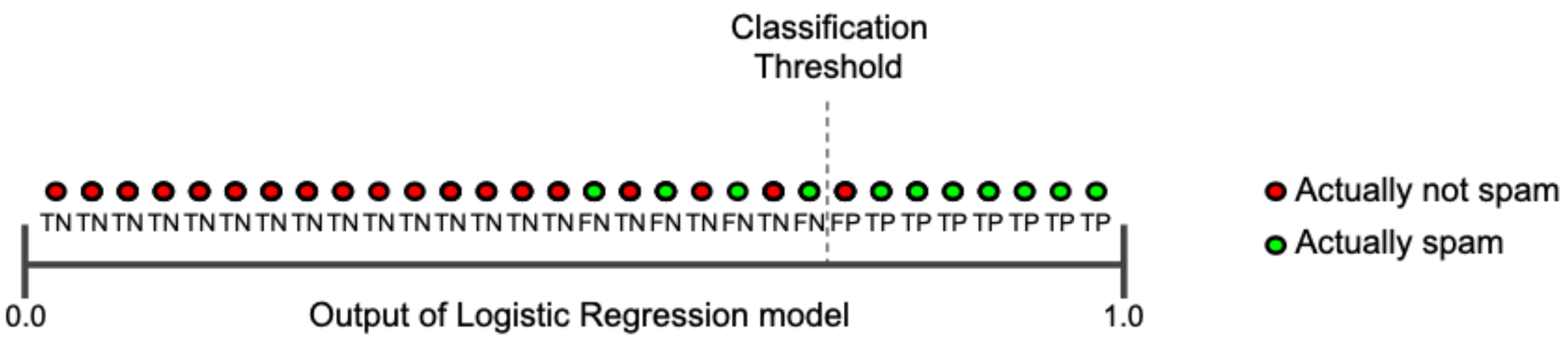
Precision vs. Recall



True Positives (TP): 8	False Positives (FP): 2
False Negatives (FN): 3	True Negatives (TN): 17

https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall

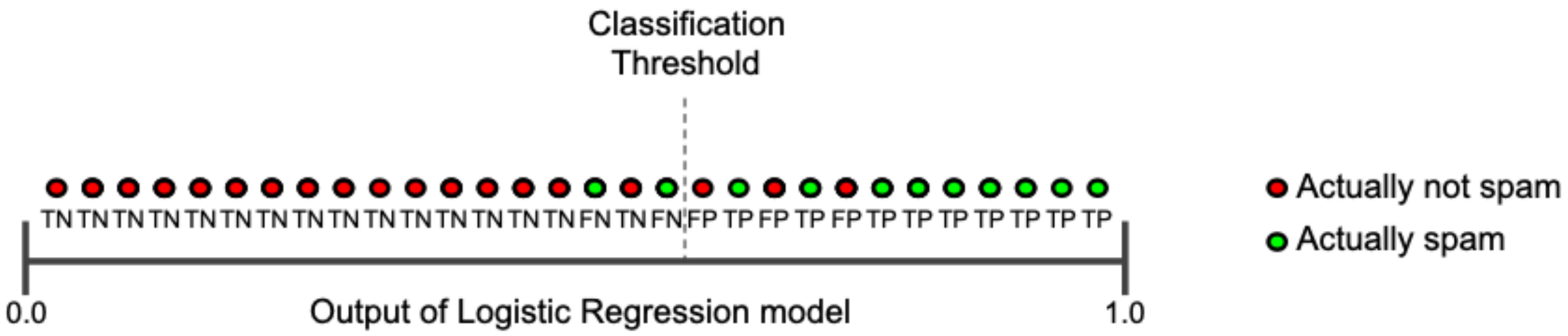
Precision vs. Recall



True Positives (TP): 7	False Positives (FP): 1
False Negatives (FN): 4	True Negatives (TN): 18

https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall

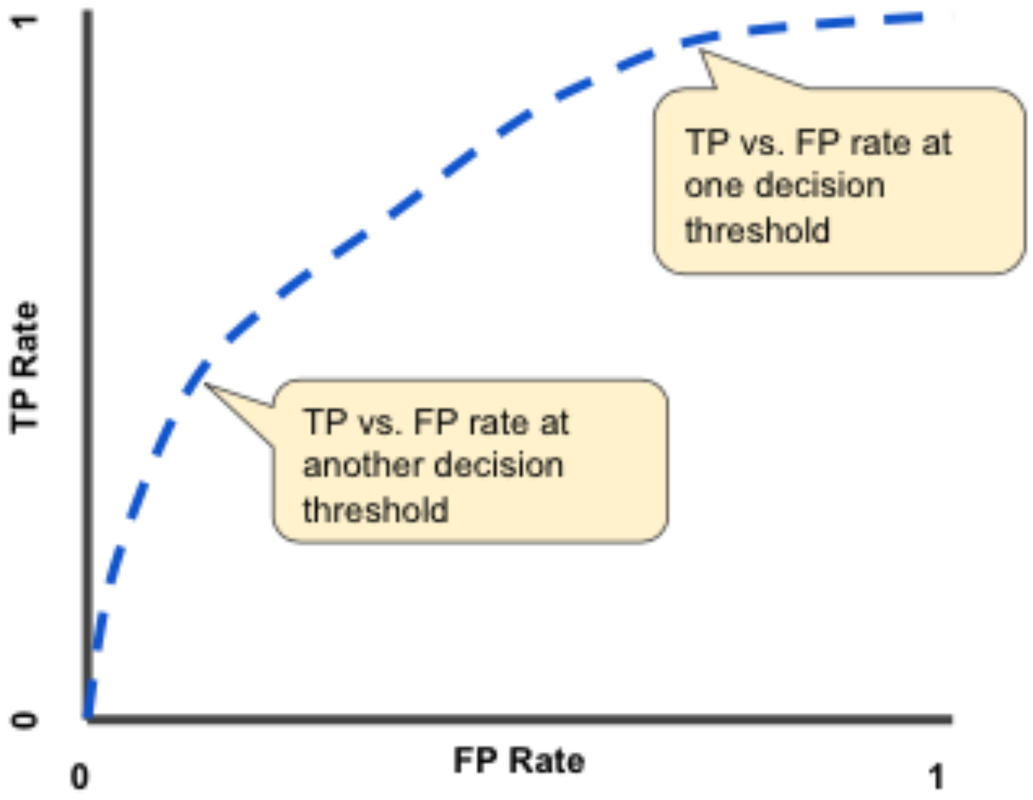
Precision vs. Recall



True Positives (TP): 9	False Positives (FP): 3
False Negatives (FN): 2	True Negatives (TN): 16

https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall

Receiver Operating Characteristic (ROC)



- Compares the True Positive Rate (TPR) on Y-axis, to the False Positive Rate (FPR) on the X- axis

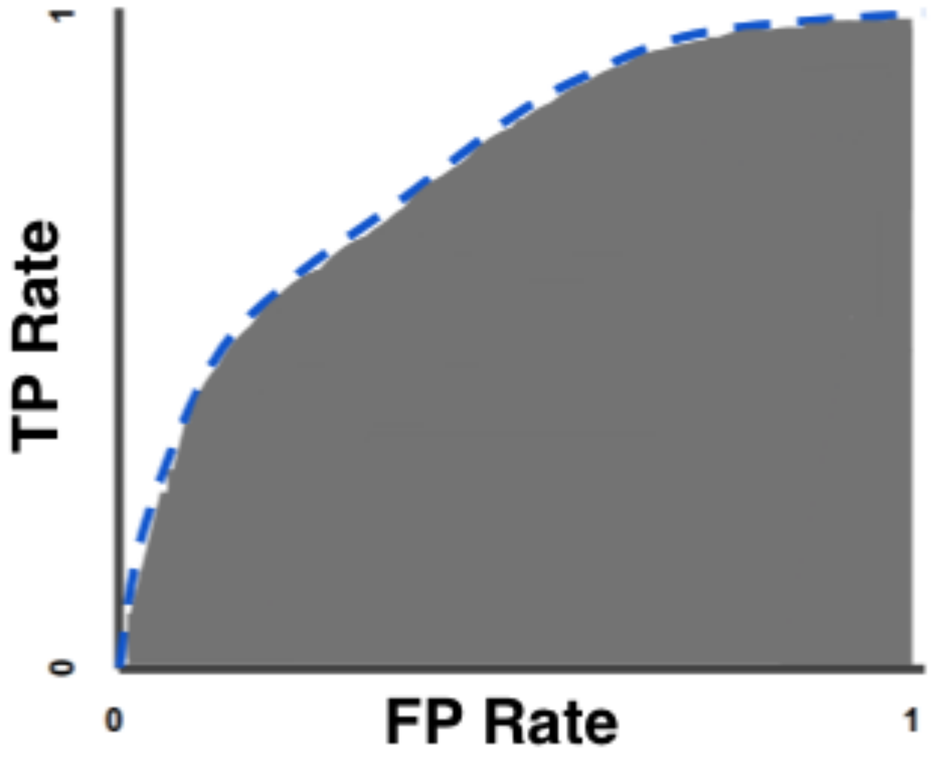
```
import matplotlib.pyplot as plt
import scikitplot as skplt

skplt.metrics.plot_roc(target_test,
predicted_probs)

plt.show()
```

https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc

Area Under Curve (AUC)

- AUC is an aggregate measure of performance across all possible classification thresholds.
- AUC ranges from 0 to 1 and a model whose predictions are 100% wrong has an AUC of 0.0; one whose predictions are 100% correct has an AUC of 1.0.

```
from sklearn.metrics import roc_auc_score

roc_auc_score(target_test, target_preds)
```



Quiz: compute metrics for each class of 3-class confusion matrix

```
target =      [1,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,2,2,2,2,1,1,2,0,0,0,0,0]
target_pred = [0,1,1,0,0,1,1,1,1,0,0,0,0,0,0,0,2,2,2,2,2,2,1,2,2,2,2,2]
print(confusion_matrix(target, target_pred))
```

True target

```
[[7 2 5]
 [3 4 2]
 [0 1 4]]
```

Predicted target

True positive	**False Negative** (Type II error)
False Positive (Type I error)	**True negative**

Accuracy = (Sum Diagonal)/ N

Precision = TP/ (TP + FP)

Recall = TP/ (TP + FN)

Quiz: compute metrics for each class of 3-class confusion matrix

```
target =      [1,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,2,2,2,2,1,1,2,0,0,0,0,0]
target_pred = [0,1,1,0,0,1,1,1,1,0,0,0,0,0,0,0,2,2,2,2,2,2,1,2,2,2,2,2]
print(confusion_matrix(target, target_pred))
```

True target

```
[[7  2  5]
 [3  4  2]
 [0  1  4]]
```

Predicted target

True positive	**False Negative** (Type II error)
False Positive (Type I error)	**True negative**

Accuracy = (7+4+4)/28 = 0.5357

Prec_Class2 = 4/(4 + 5 + 2) = 0.3636

Rec_Class2 = 4/(4 + 1 + 0) = 0.8

…

```
Precision:  [ 0.7  0.57142857  0.36363636]
Recall:  [ 0.5  0.44444444  0.8]
```

Where is the biggest crime scene?

Confusion matrices all with equal accuracy 0.6875!!!
How about precision and recall?

```
[[10  0]
 [5  1]]
```

A

```
[[7  2]
 [3  4]]
```

B

```
[[0  4]
 [1  11]]
```

C

Where is the biggest crime scene?

Confusion matrices all with equal accuracy 0.6875!!!

How about precision and recall?

```
Precision:  [ 0.7  1]
Recall:  [ 1  0.2]
```

```
[[10  0]
 [5  1]]
```

A

```
Precision:  [ 0.7  0.7]
Recall:  [ 0.8  0.6]
```

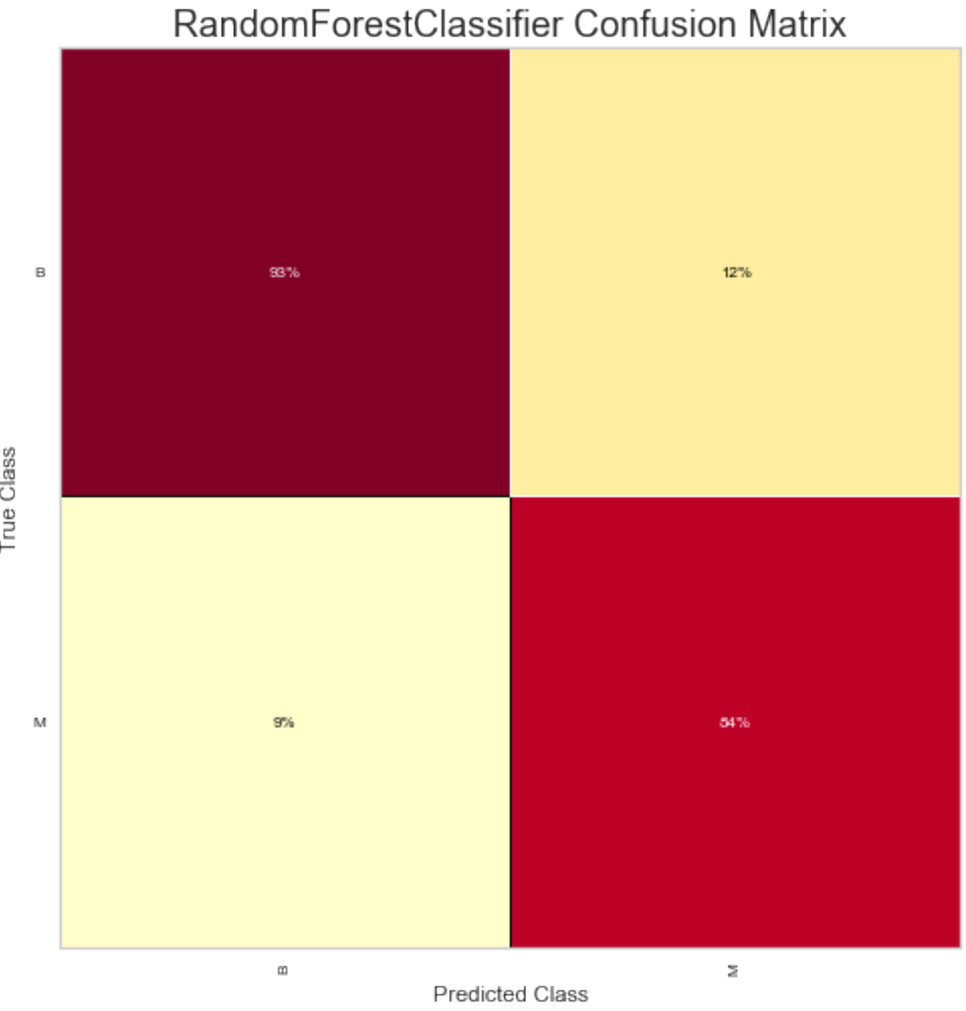
```
[[7  2]
 [3  4]]
```

B

```
Precision:  [ 0  0.7]
Recall:  [ 0  0.9]
```

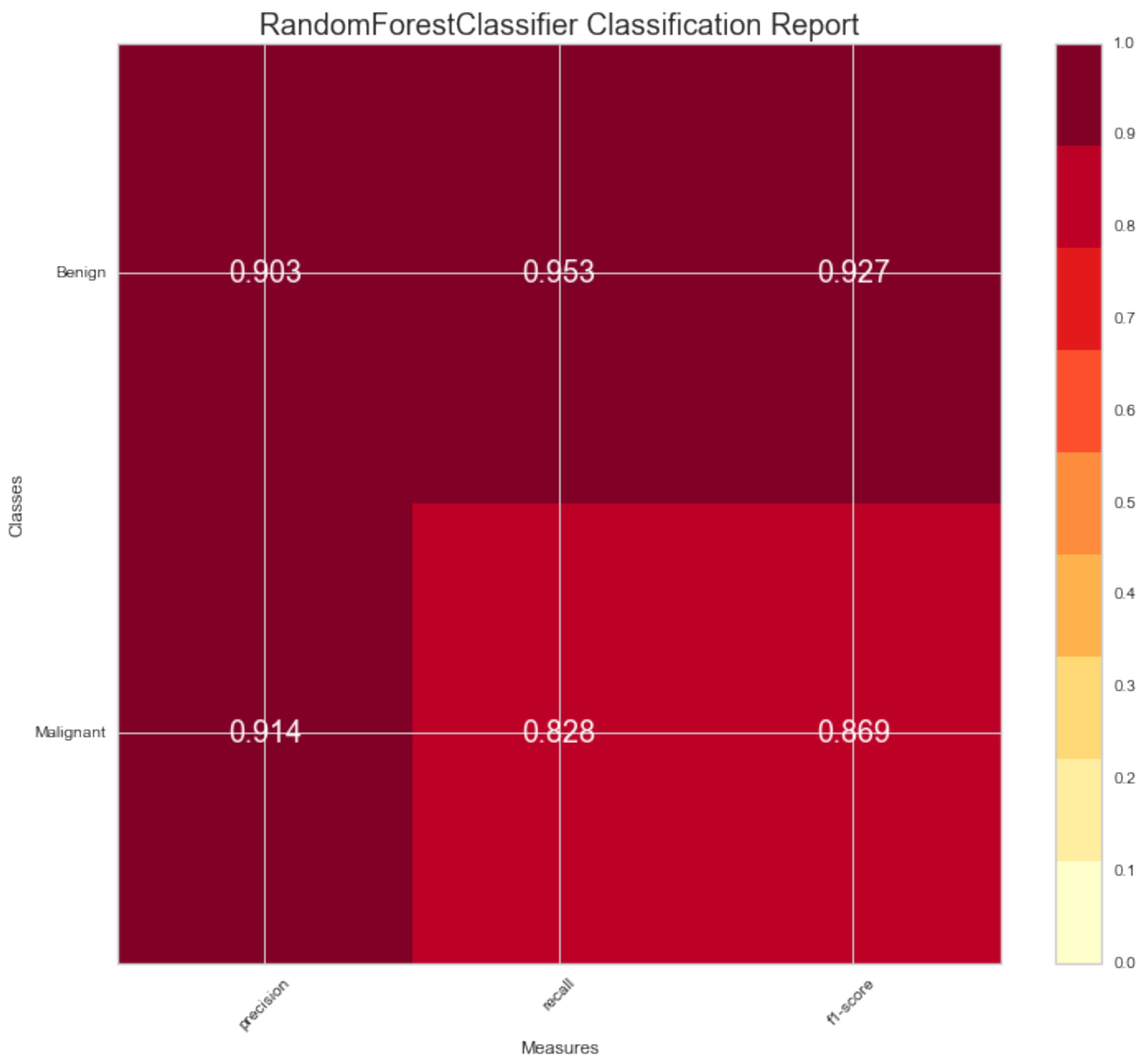
```
[[0  4]
 [1  11]]
```

Visualizing the Confusion Matrix

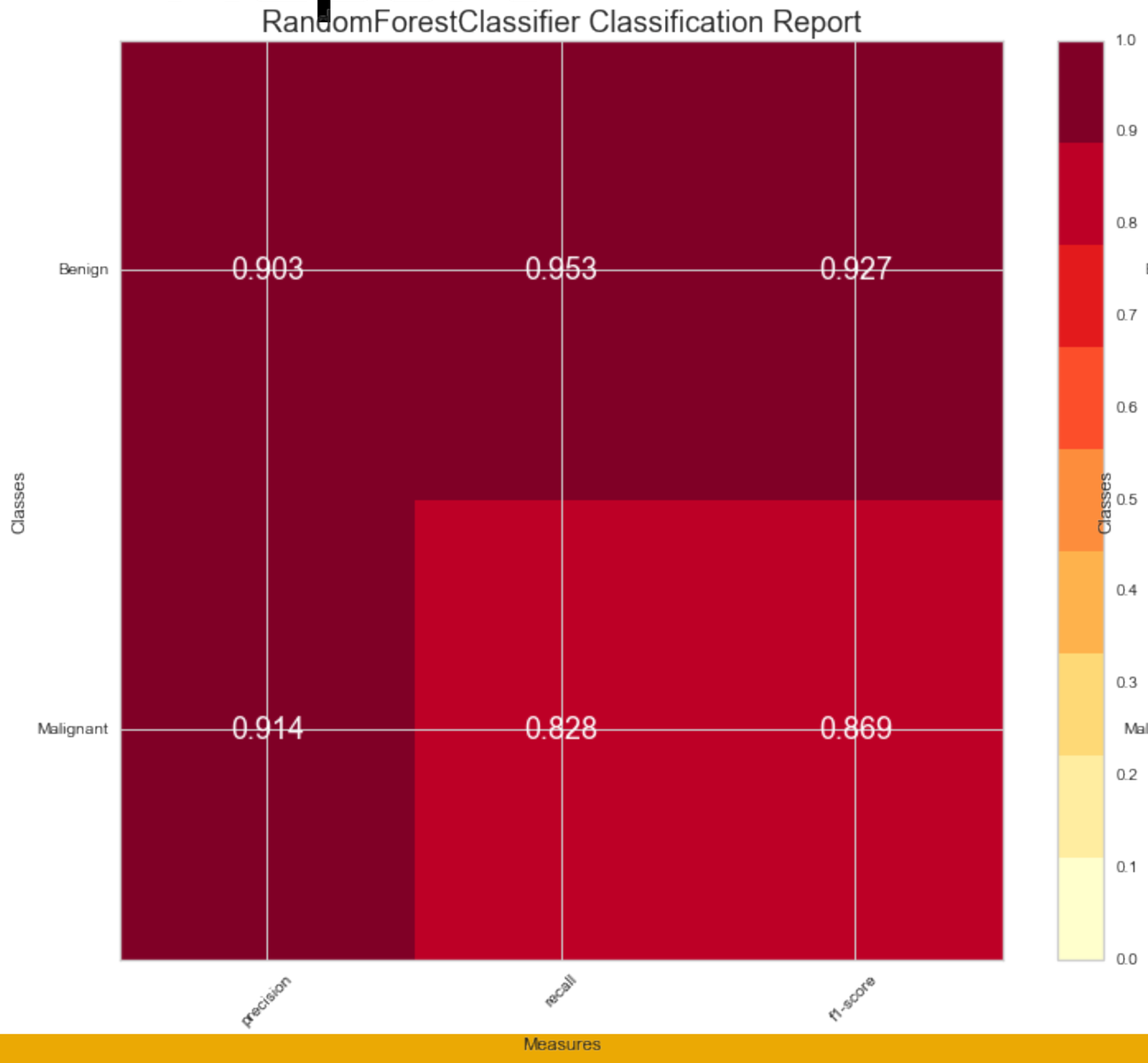


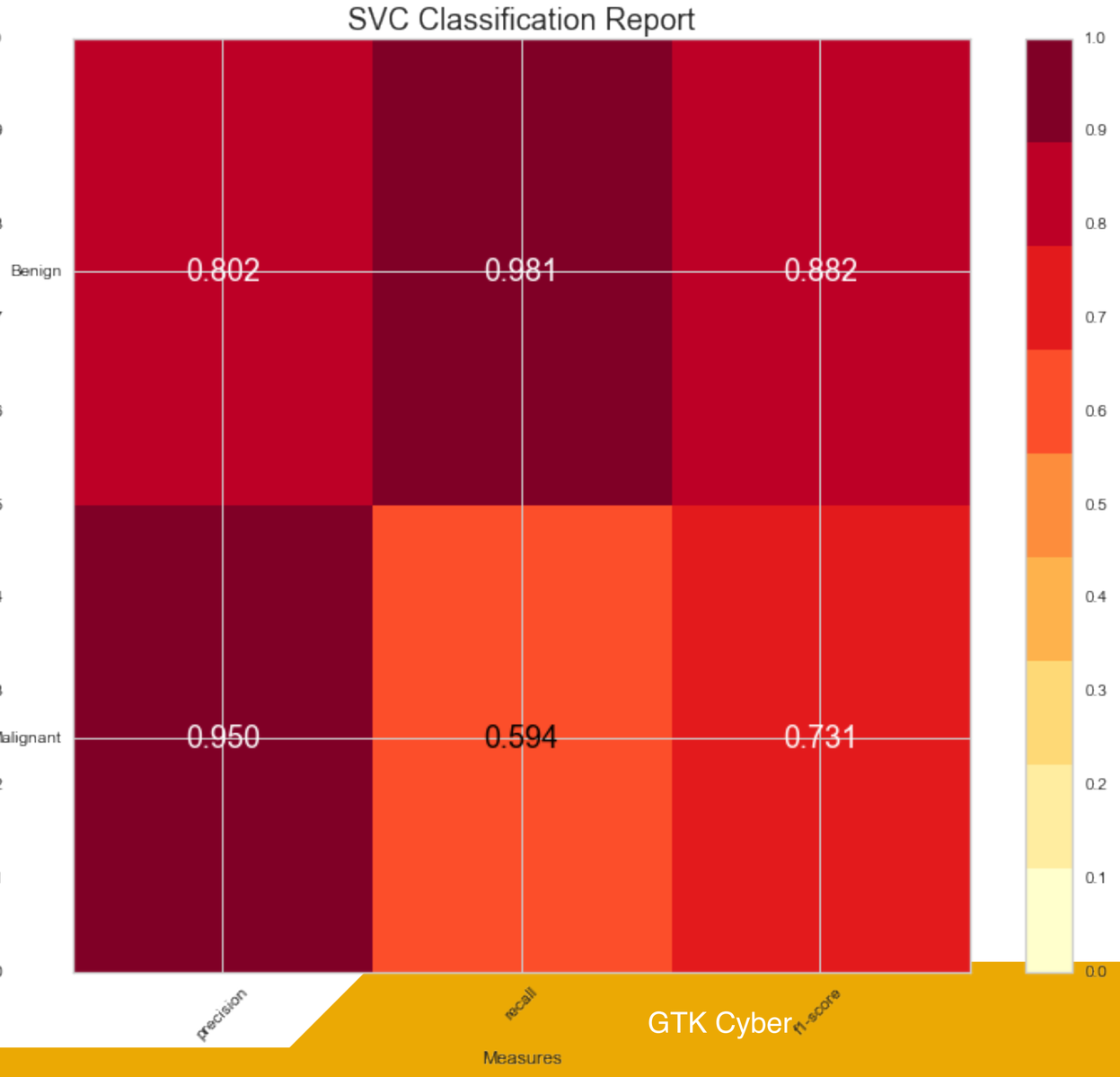

```
from yellowbrick.classifier import ConfusionMatrix

#Create the models for both the Random Forest
random_forest_model = RandomForestClassifier()

random_forest_conf_matrix = ConfusionMatrix(
                                random_forest_model )
random_forest_conf_matrix.fit(features_train, target_train)
random_forest_conf_matrix.score(features_test, target_test)
random_forest_conf_matrix.poof()
```

Visualizing the Classification Report

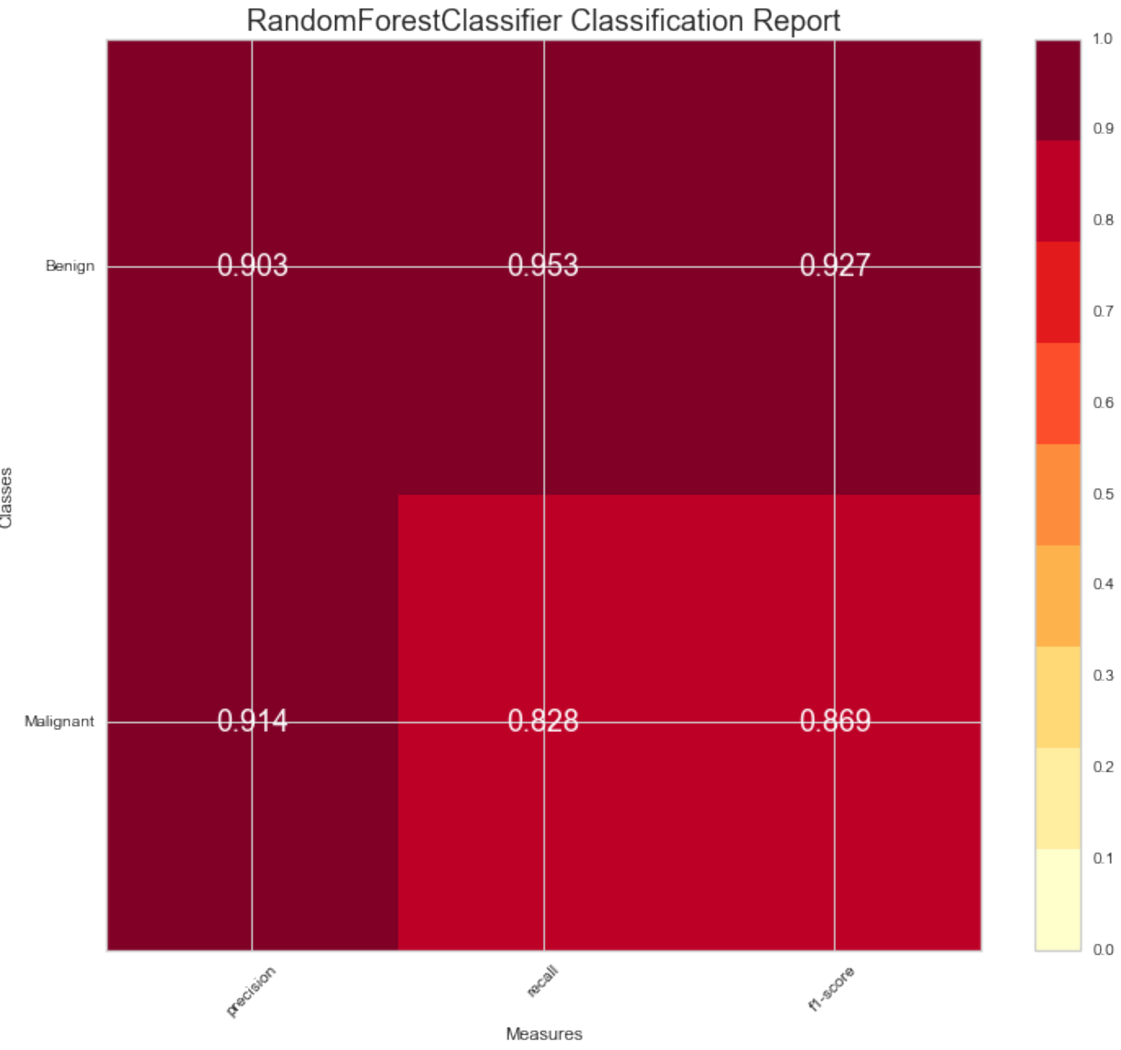


Visualizing the Classification Report





Visualizing the Classification Report




```
from yellowbrick.classifier import ClassificationReport

random_forest_class_report = ClassificationReport( random_forest_model,
                                                   classes=['Benign', 'Malignant'])
random_forest_class_report.fit(features_train, target_train)
random_forest_class_report.score(features_test, target_test)
random_forest_class_report.poof()
```

Cross-Validation

Why cross-validation?

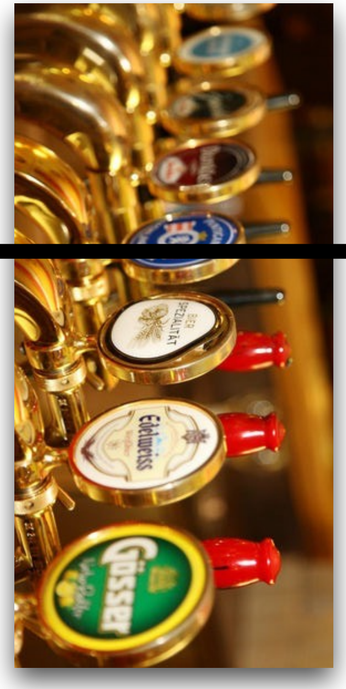
Because you don’t wanna be that guy!!!!

Split data for train and test

features(X)

target (y)

70%



Simple train test split!

```
# Simple Cross-Validation: Split the data set into training and test data

X_train, X_test, target_train, target_test =
model_selection.train_test_split(X, target, test_size=0.25)
```

Simple cross-validation!

```
## Train the classifier
clf = tree.DecisionTreeClassifier()
clf = clf.fit(features_train, target_train)

## Making predictions using test data
target_pred = clf.predict(features_test)

## Report metrics
accuracy = metrics.accuracy_score(target_test, target_pred)
```

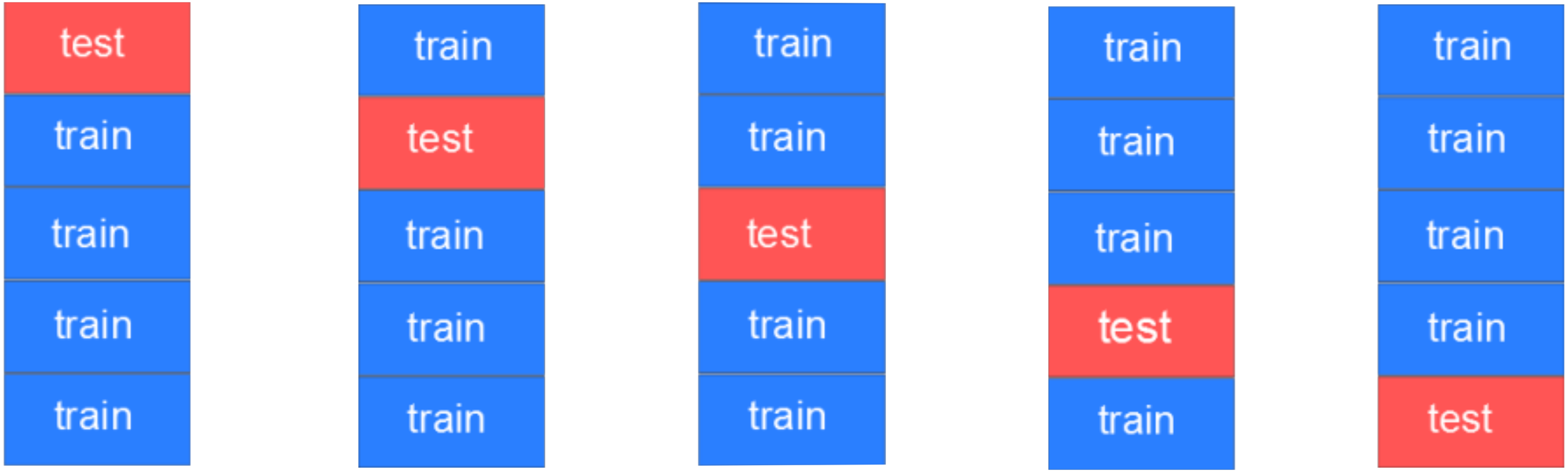
Cross Validation

Better approach: Create a bunch of train/test splits, calculate the testing accuracy for each, and average the results together.

Cross Validation

1. Split the dataset into K **equal** partitions (or "folds").
2. Use fold 1 as the **testing set** and the union of the other folds as the **training set**.
3. Calculate testing accuracy.
4. Repeat steps 2 and 3 *K* times, using a **different fold** as the testing set each time.
5. Use the **average testing accuracy** as the estimate of out-of-sample accuracy.

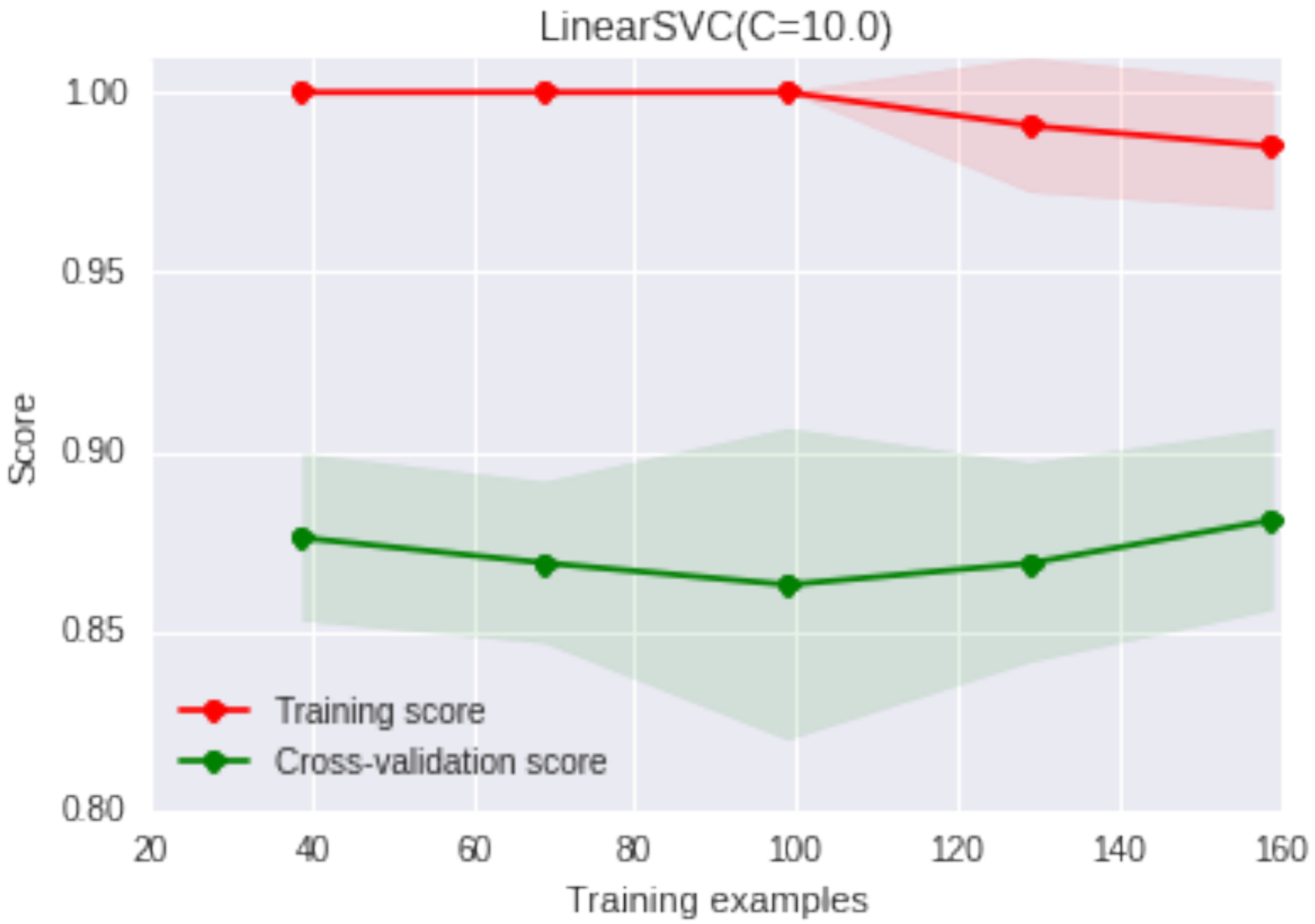
K-Fold Cross Validation



K-fold Cross-Validation

```
scores = cross_val_score(clf, features, target, cv=<n>)
scores.mean()
```

Visualizing Goodness of Fit



https://jmetzen.github.io/2015-01-29/ml_advice.html

Visualizing Goodness of Fit

```
from yellowbrick.classifier.learning_curve import LearningCurveVisualizer


viz = LearningCurveVisualizer(GaussianNB())
viz.fit(features,target)
viz.poof()
```

Visualizing Goodness of Fit

```
from yellowbrick.classifier.learning_curve import LearningCurveVisualizer
viz = LearningCurveVisualizer(GaussianNB())
viz.fit(features,target)
viz.poof()
```

https://jmetzen.github.io/2015-01-29/ml_advice.html

In Class Exercise

Please take 45 minutes and complete

Worksheet 5.2 - Malicious URL Classification

Future Topics

- Save (e.g. pickle) models
- Feature Importances
- Bag-of-Words methods result in a sparse matrix
- Is accuracy of 0.86 acceptable for production? NO! Will result in high false positive rate!
- Much more data and/or better data, tweaking feature engineering/selection and model selection needed
- Consider Deep Learning with word2vec embedding!

Feature Importances

0.524576	DurationCreated
0.092815	Length
0.080639	EntropyDomain
0.059617	DigitsCount
0.057386	FirstDigitIndex
0.057095	LengthDomain
0.008454	com
0.006945	net
0.005946	news.1
0.005724	org
0.004589	ru

Hyper-Parameter Tuning!

General Optimization: e.g. Gradient Descent!

Grid Search

```
RandomForestClassifier(bootstrap=True,
class_weight=None,
criterion='gini',
max_depth=None,
max_features='auto',
max_leaf_nodes=None,
min_impurity_decrease=0.0,
min_impurity_split=None,
min_samples_leaf=1,
min_samples_split=2,
min_weight_fraction_leaf=0.0,
n_estimators=10,
n_jobs=1,
oob_score=False
)
```

Tuning these parameters

- GridSearchCV: You provide a list of possible parameters
- RandomizedSearchCV: Random combinations are searched

Grid Search

```
param_grid = [
  {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
  {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
'kernel': ['rbf']},
 ]
```

Grid Search

```
# use a full grid over all parameters
param_grid = {"max_depth": [3, None],
              "max_features": [1, 3, 10],
              "min_samples_split": [2, 3, 10],
              "min_samples_leaf": [1, 3, 10],
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"]}

# run grid search
grid_search = GridSearchCV(clf, param_grid=param_grid)
start = time()
grid_search.fit(features, target)

print("GridSearchCV took %.2f seconds for %d candidate parameter settings."
      % (time() - start, len(grid_search.cv_results_['params'])))
report(grid_search.cv_results_)
```

http://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html

Random Search

```
# specify parameters and distributions to sample from
param_dist = {"max_depth": [3, None],
              "max_features": sp_randint(1, 11),
              "min_samples_split": sp_randint(2, 11),
              "min_samples_leaf": sp_randint(1, 11),
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"]}

# run randomized search
n_iter_search = 20
random_search = RandomizedSearchCV(clf, param_distributions=param_dist,
                                   n_iter=n_iter_search)

start = time()
random_search.fit(features, target)
print("RandomizedSearchCV took %.2f seconds for %d candidates"
      " parameter settings." % ((time() - start), n_iter_search))
report(random_search.cv_results_)
```

http://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html

Bayesian Search

```
pip install scikit-optimize
```

```
# run bayesian-optimized hyperparameter search
n_iter_search = 20
bayes_search = BayesSearchCV(clf,
param_distributions=param_dist,
                                    n_iter=n_iter_search)

start = time()
bayes_search.fit(features, target)
print("BayesSearchCV took %.2f seconds for %d candidates"
      " parameter settings." % ((time() - start),
n_iter_search))
report(bayes_search.cv_results_)
```

https://scikit-optimize.github.io/notebooks/sklearn-gridsearchcv-replacement.html

Pipelines

Without Pipeline

```
# get categorical features
# drop off last column because its unnecessary
X_categorical = pd.get_dummies(df, columns=categorical_columns)\
                 .astype(int).iloc[:,:-1]

# get and transform numeric features
X_numeric = df[numeric_columns]
X_numeric[numeric_columns] =
  StandardScaler().fit_transform(X_numeric)

# get outcome variable
y = df[target]

# combine transformed categorical and numeric features
X_final = pd.concat((X_numeric, X_categorical), axis=1)
```

Without Pipeline

```
# create rf regressor and check 10-fold RMSE
rf = RandomForestRegressor()
cross_val_scores = np.abs(cross_val_score(
  rf, X_final, y, cv=10
  scoring="neg_mean_squared_error"
))
rmse_cross_val_scores = np.sqrt(cross_val_scores)
```

With Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScalar
from sklearn.neighbors import KNeighborsClassifier

# it takes a list of tuples as parameter
pipeline = Pipeline([
    ('scaler',StandardScaler()),
    ('clf', KNeighborsClassifier())
])

pipeline.fit(X_train,y_train)
```

Pipelines

```
from sklearn.feature_selection import SelectKBest
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline

steps = [('feature_selection', SelectKBest(k=100)),
         ('random_forest', RandomForestClassifier())]

pipeline = Pipeline(steps)
```

With Pipeline

full_pipeline.steps

```
[('all_features', FeatureUnion(n_jobs=1, transformer_list=[
   ('categoricals', Pipeline(memory=None, steps=[
     ('selector', ItemSelector(key=['rbc', 'pc', 'pcc', 'ba', 'htn', 'dm',
                                    'cad', 'appet', 'pe', 'ane'])),
     ('imputer', Imputer(axis=0, copy=True, missing_values=0,
                         strategy='most_frequent', verbose=0)),
     ('encoder', OneHotEncoder(cat ... tegy='median', verbose=0)),
     ('scaler', StandardScaler(copy=True, with_mean=True, with_std=True))]
   ))
 ], transformer_weights=None)),
 (‘rf_classifier', RandomForestClassifier(bootstrap=True, class_weight=None,
             criterion=‘gini', max_depth=None, max_features=‘auto',
             max_leaf_nodes=None, min_impurity_decrease=0.0,
             min_impurity_split=None, min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
             oob_score=False, random_state=None, verbose=0,
             warm_start=False))]
```

Pipelines

```
pipeline.fit( features_train, target_train )

y_prediction = pipeline.predict( features_test )

report = classification_report( target_test, target_prediction

print(report)
```

With Pipeline

```
cross_val_scores = np.abs(cross_val_score(
  full_pipeline, X, y, cv=10,
  scoring="neg_mean_squared_error"
))
rmse_cross_val_scores = np.sqrt(cross_val_scores)
```

Why use a pipeline?

- It makes code more readable
- You don’t have to worry about keeping track data during intermediate steps, for example between transforming and estimating.
- It makes it trivial to move ordering of the pipeline pieces, or to swap pieces in and out.
- It allows you to do GridSearchCV on your workflow

Pickling Your Model

- Pickling your model allows you to preserve your model to be used later.

Pickling Your Model

In order to rebuild a similar model with future versions of scikit-learn, additional metadata should be saved along the pickled model:

- The training data, e.g. a reference to a immutable snapshot
- The python source code used to generate the model
- The versions of scikit-learn and its dependencies
- The cross validation score obtained on the training data

Pickling Your Model

```
#Saving your model
from sklearn.externals import joblib
joblib.dump(clf, 'filename.pkl')

#Loading your model
clf = joblib.load('filename.pkl')
```

Another Option: Scikit-Deploy

- Scikit-Deploy automatically creates a docker container with a REST API for your model
- This container can be easily deployed on any server with docker running.
- Docs available here: https://pypi.org/project/scikit-deploy/

Another Option: Scikit-Deploy

- Scikit-Deploy automatically creates a docker container with a REST API for your pickled model
- This container can be easily deployed on any server with docker running.
- Once running, simply issue a POST request with your features, and the model will return predictions
- Docs available here: https://pypi.org/project/scikit-deploy/

In Class Exercise

Please take 30 minutes and complete

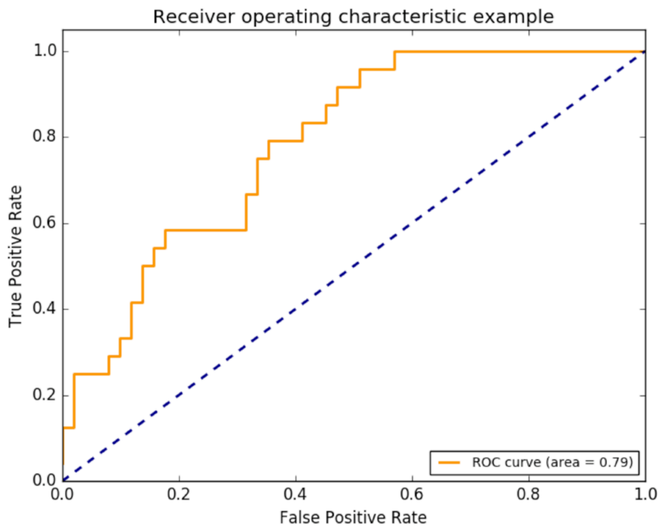
Worksheet 5.3: Optimizing your Model

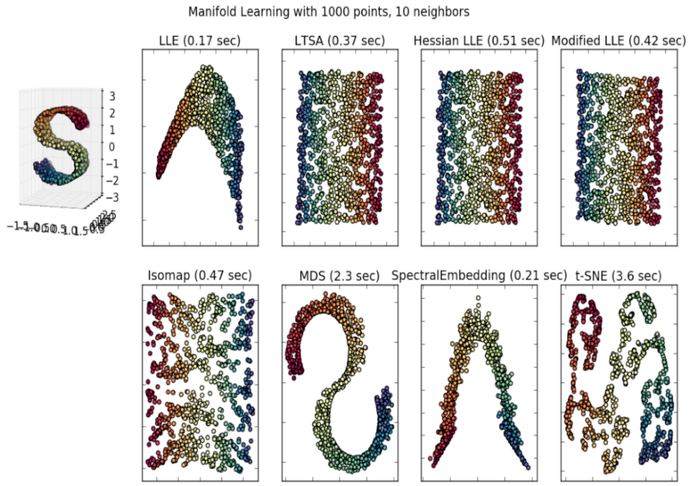
Custom Scorer Metrics Function!

```
def prec_recall_min_scorer(target_true, target_pred):
        # type: score_func(target_true, target_pred, **kwargs)
        precision = metrics.precision_score(target_true, target_pred, average=None)
        recall = metrics.recall_score(target_true, target_pred, average=None)
        return min(min(precision), min(recall))

custom_score = metrics.make_scorer(prec_recall_min_scorer, greater_is_better=True)

# For example
grid_search = model_selection.RandomizedSearchCV(clf, param_distributions=param_dist,
                                                 cv=sss.split(X, target),
                                                 n_iter=n_iter_search,
                                                 # using custom scorer defined above
                                                 # "maximize the min of precision and recall"
                                                 scoring=custom_score())
```

Future Studies - sklearn

- Feature Selection using Classifiers
- Ranking of best features
- Compare multiple Classifiers and pick best!
- Model Stacking or Majority Voting
- Plotting (ROC curve, manifold learning)



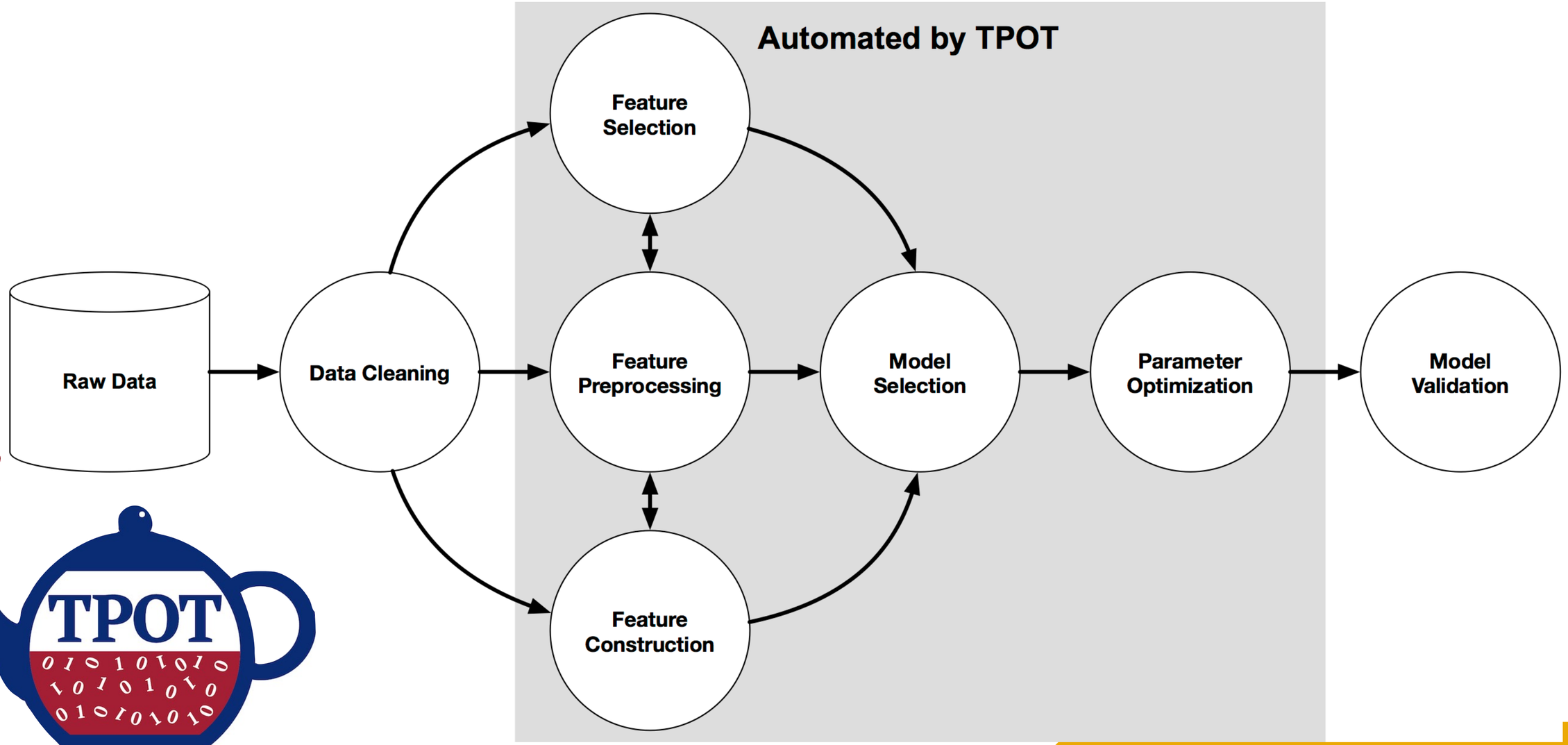


What else can you automate?

What else can you automate? It turns out, almost everything with TPOT.

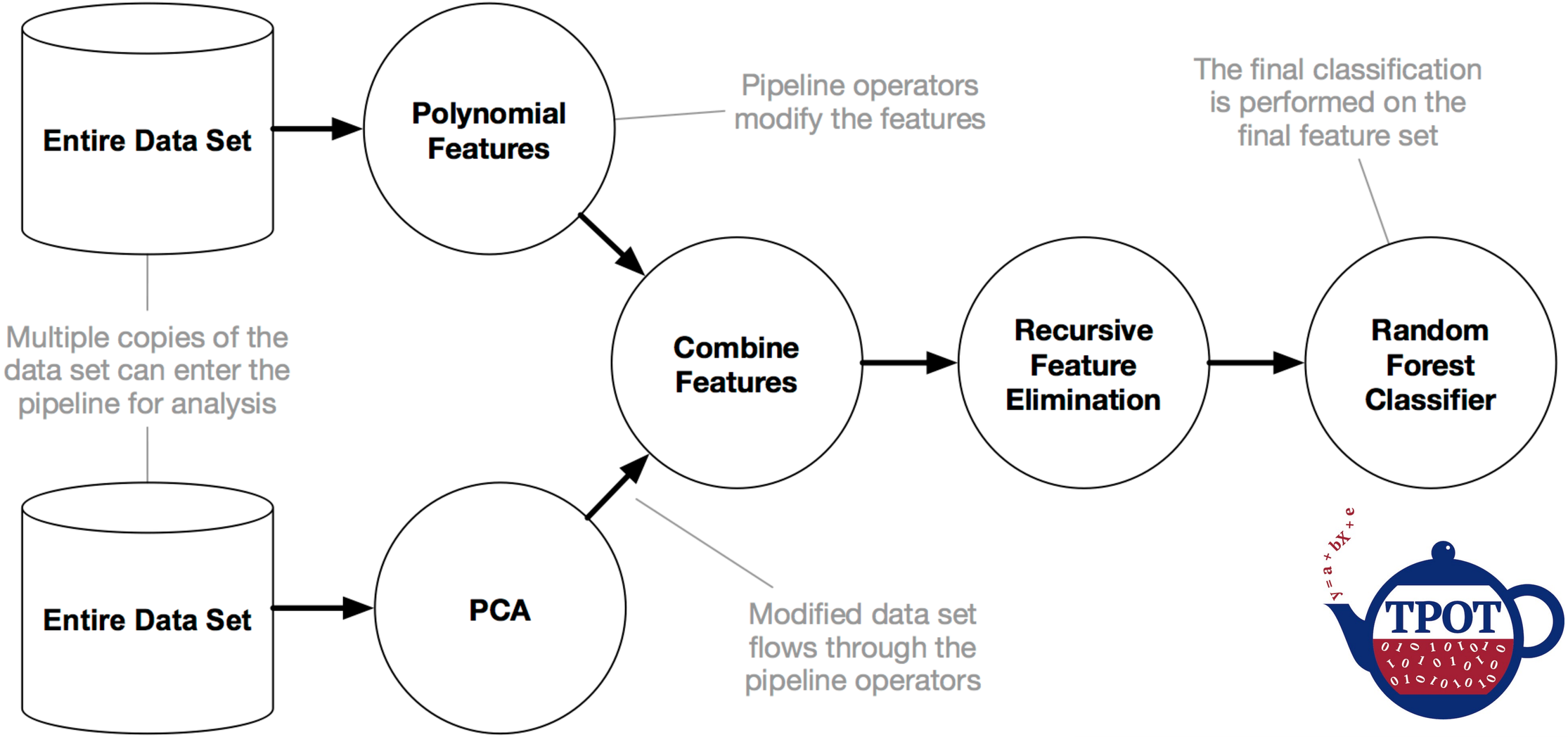
TPOT will automate the most tedious part of machine learning by intelligently exploring thousands of possible pipelines to find the best one for your data.

Auto ML



https://github.com/EpistasisLab/tpot

Auto ML



https://github.com/EpistasisLab/tpot

Auto ML

```
from tpot import TPOTClassifier

tpot = TPOTClassifier(generations=5, population_size=20, verbosity=2)
tpot.fit(features_train, target_train)
print(tpot.score(features_test, target_test))
tpot.export('tpot_pipeline.py')
```

https://github.com/EpistasisLab/tpot

Auto ML

Docs available here: http://epistasislab.github.io/tpot/

https://github.com/EpistasisLab/tpot

Questions?

In Class Exercise

Please take 30 minutes and complete

Worksheet 5.4: Automate it all!