DATA SOCIETY:

Advanced Classification - Part 3

One should look for what is and not what he thinks should be. (Albert Einstein)

Module completion checklist

Objective	Complete
Explain metrics used to assess binary classifier performance	
Evaluate base random forest model	
Optimize random forest model using RandomizedCV method	
Use performance metrics to compare optimized RF to base RF model	

Loading packages

- Let’s load the packages we will be using today

```
# Helper packages.
import os
import pickle
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import math
from pathlib import Path

# Scikit-learn packages for building models and model evaluation.
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn import metrics
```

Recap: Random Forests

What is Random Forests?

- Ensemble method used for **classification and regression tasks**
- Supervised learning algorithm which builds **multiple decision trees** and aggregates the result
- Uses a technique called Bootstrap Aggregation, commonly known as **Bagging**

Recap: why Random Forests?

- Consistently high predictive accuracy for a wide range of problems
- Efficiency with large datasets, especially those with a lot of variables
- Reduced bias error
- Resistance to correlation between variables
- (Some) resistance to overfitting

Random Forests methodology overview

Step 1:
Set *n* & *m* for forest

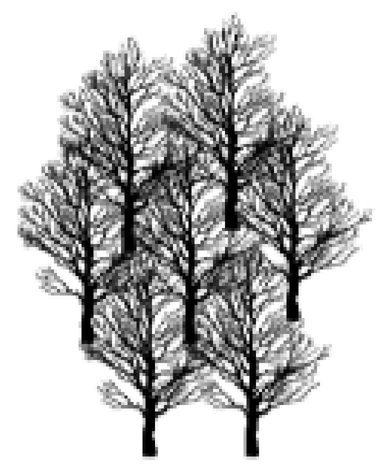
Step 2:
Build forest on training data

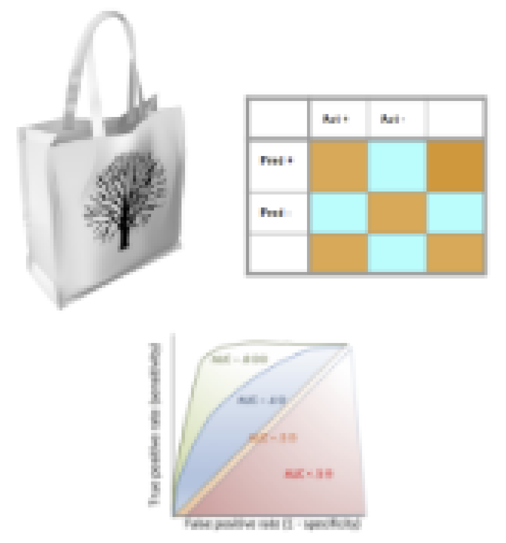
Step 3:
Check performance

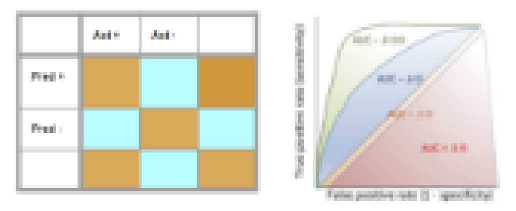
Step 4:
Apply to test data & evaluate

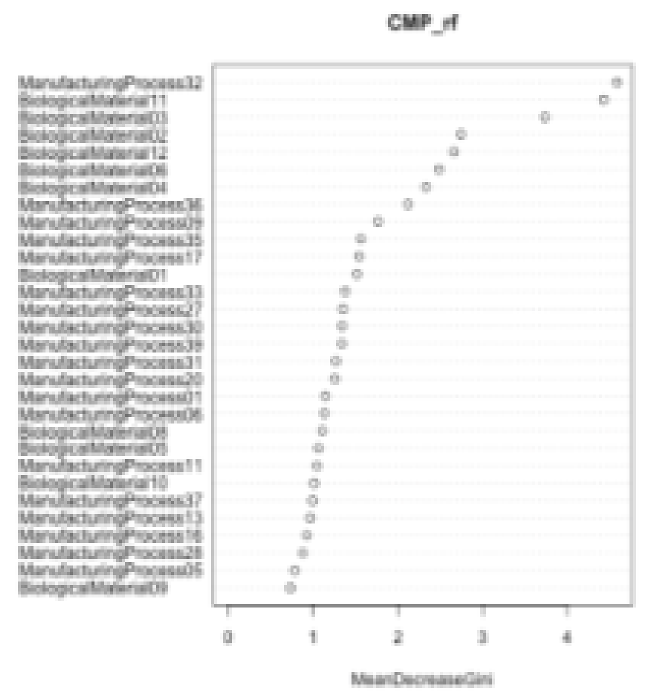
Step 5:
Use variable importance as needed

How many trees?
How many variables do we get to pick?









Review data cleaning steps

- **Today, we will be loading the cleaned dataset we used in last module**
- To recap, the steps to get to this cleaned dataset were:
  - Remove household ID and individual ID
  - Remove variables with over 50% NAs
  - Transformed target variable to binary

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- We will use the `pathlib` library
- Let the `main_dir` be the variable corresponding to your `course` folder
- Let `data_dir` be the variable corresponding to your `data` folder

```
# Set 'main_dir' to location of the project folder
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

Load the cleaned dataset

- Let's load the dataset from last module: `costa_clean`
- Assign it to `costa_clean` variable

```
costa_clean = pickle.load(open("costa_clean.sav","rb"))
print(costa_clean.head())
```

```
   rooms  tablet  males_under_12  ...  rural_zone  age  Target
0      3       0               0  ...           0   43    True
1      4       1               0  ...           0   67    True
2      8       0               0  ...           0   92    True
3      5       1               0  ...           0   17    True
4      5       1               0  ...           0   37    True

[5 rows x 81 columns]
```

Print info for our data

- Let's view the column names

```
costa_clean.columns
```

```
Index(['rooms', 'tablet', 'males_under_12', 'males_over_12', 'males_tot',
       'females_under_12', 'females_over_12', 'females_tot', 'ppl_under_12',
       'ppl_over_12', 'ppl_total', 'years_of_schooling', 'wall_block_brick',
       'wall_socket', 'wall_prefab_cement', 'wall_wood', 'floor_mos_cer_terr',
       'floor_cement', 'floor_wood', 'ceiling', 'electric_public',
       'electric_coop', 'toilet_sewer', 'toilet_septic', 'cookenergy_elec',
       'cookenergy_gas', 'trash_truck', 'trash_burn', 'wall_bad', 'wall_reg',
       'wall_good', 'roof_bad', 'roof_reg', 'roof_good', 'floor_bad',
       'floor_reg', 'floor_good', 'disabled_ppl', 'male', 'female', 'under10',
       'free', 'married', 'separated', 'single', 'hh_head', 'hh_spouse',
       'hh_child', 'num_child', 'num_adults', 'num_65plus', 'num_hh_total',
       'dependency_rate', 'male_hh_head_educ', 'female_hh_head_educ',
       'meaneduc', 'educ_none', 'educ_primary_inc', 'educ_primary',
       'educ_secondary_inc', 'educ_secondary', 'educ_undergrad', 'bedrooms',
       'ppl_per_room', 'house_owned_full', 'house_owned_paying',
       'house_rented', 'house_other', 'computer', 'television',
       'num_mobilephones', 'region_central', 'region_Chorotega',
       'region_pacifico', 'region_brunca', 'region_antlantica',
       'region_huetar', 'urban_zone', 'rural_zone', 'age', 'Target']
```

Split into training and test sets

```
# Select the predictors and target.
X = costa_clean.drop(['Target'], axis = 1)
y = np.array(costa_clean['Target'])

# Set the seed to 1.
np.random.seed(1)

# Split into training and test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)
```

Recap: fit vanilla RF model

- Let's build a vanilla (a.k.a. pure and simple) Random Forests model to serve as our baseline

```
# Initialize the classifier.
forest = RandomForestClassifier()

# Fit training data.
forest.fit(X_train, y_train)
```

```
RandomForestClassifier()
```

Recap: predict using vanilla RF model

```
# Predict on test.
forest_y_predict = forest.predict(X_test)
print(forest_y_predict[:5])
```

```
[ True  True  True  True  True]
```

```
#Predict on test, but instead of labels
# we will get probabilities for class 0 and 1.
forest_y_predict_prob = forest.predict_proba(X_test)
print(forest_y_predict_prob[5:])
```

```
[[0.74 0.26]
 [0.01 0.99]
 [0.02 0.98]
 ...
 [0.07 0.93]
 [0.23 0.77]
 [0.75 0.25]]
```

Recap: confusion matrix and accuracy

	Positive	Negative
Positive	TP	FP
Negative	FN	TN

- $ACC = \frac{(TP+TN)}{(TP+FP+FN+TN)}$

```
# Compute accuracy for RF model.
forest_accuracy = metrics.accuracy_score(y_test,
                                          forest_y_predict)
print(forest_accuracy)
```

```
0.9483960948396095
```

- **The most commonly used** metric for classification problems
- **The most intuitive**: the ratio of correct predictions to total cases
- Fantastic and **easy to interpret** metric for well-balanced datasets

When is accuracy not so accurate

- *Why not just use accuracy?*
  - Because it takes both true positives and true negatives into account, but doesn't tell us how many of either we had
  - Bad for **class-imbalanced datasets**
  - Say a bank needs to tag a transaction as fraudulent, if the algorithm says No 100% of the time it will be accurate about 99% of the time, is this a good model though?

- *What are other metrics you know?*
  - Precision
  - Recall
  - F1 score
  - ...

Precision

	Positive	Negative
Positive	TP	FP
Negative	FN	TN

```
forest_precision = metrics.precision_score(y_test,
                                           forest_y_predict)
print(forest_precision)
```

```
0.9447424892703863
```

- $PR = \frac{(TP)}{(TP+FP)}$
- A proportion of values that is truly positive out of all predicted positive values
- A.K.A. PPV - positive predicted value

- In our fraud detection example, $PR = 0$, thus making our seemingly accurate algorithm a total fraud *(pun intended)*
- Great metric to use **when we want to be very sure of our prediction**, *think convicting a person who is truly guilty*
- The flip side in being very precise is *letting some criminals walk free* or **catching too many false negatives**

Recall

	Positive	Negative
Positive	TP	FP
Negative	FN	TN

```
forest_recall = metrics.recall_score(y_test,
                                     forest_y_predict)
print(forest_recall)
```

```
0.9750830564784053
```

- $RE = \frac{(TP)}{(TP+FN)}$
- Proportion of actual positives that is classified correctly
- A.K.A. sensitivity, hit rate, or true positive rate (TPR)

- In the fraud detection case, our algorithm failed to mark a single positive value, hence $RE = 0$ in this case too -> *fail*
- Great metric to use **when we want to capture as many positives as possible**, e.g. *fraud detection*
- Recall is 1 if we mark every single transaction as fraudulent, though, so the flip side is **catching too many false positives**

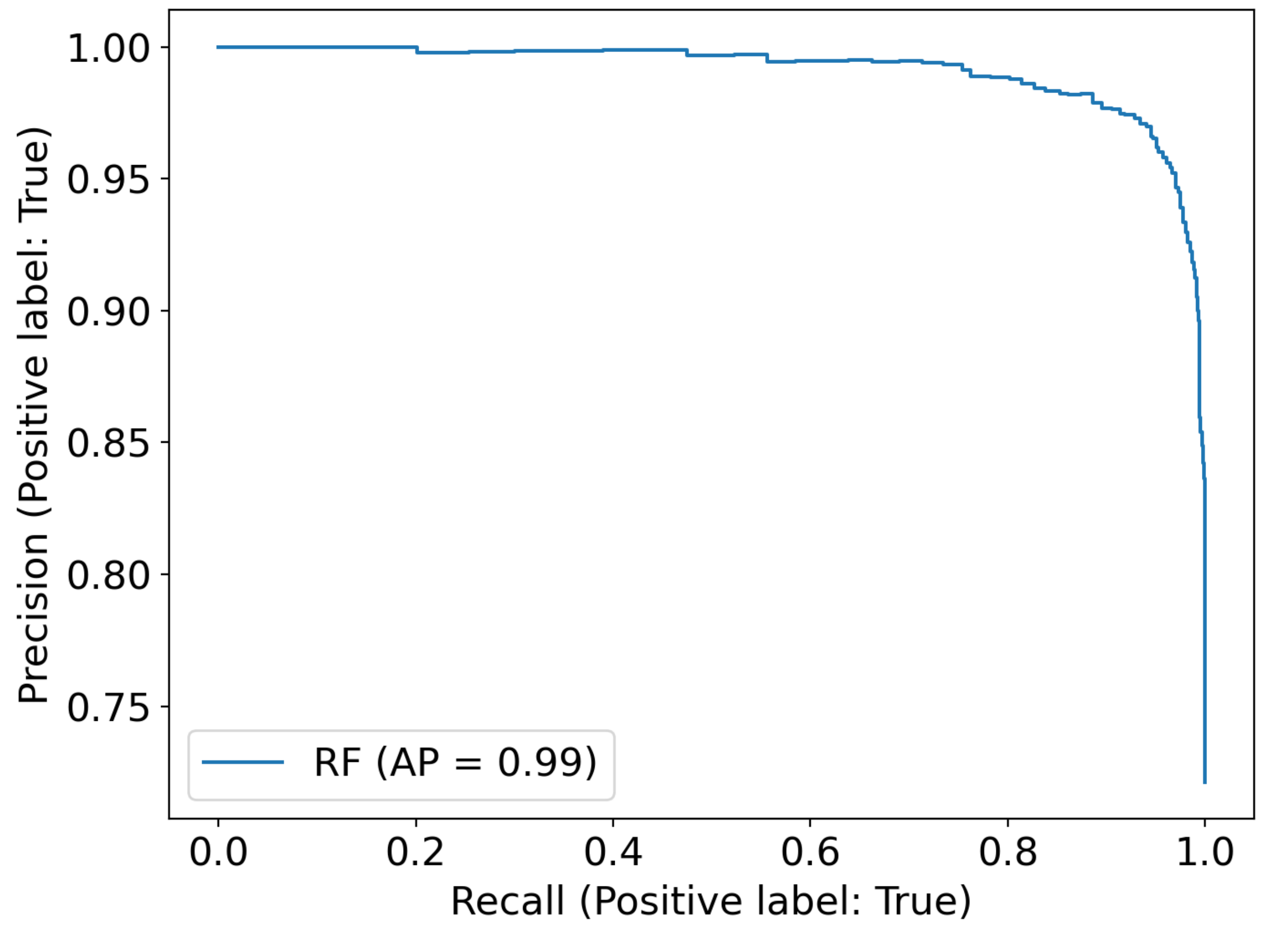
Precision-recall curve: the tradeoff

- It often helps to visualize the tradeoff between precision and recall
- That's why there is a plot known as **precision-recall** curve
- It is plotted
  - For a range of possible values of the probability thresholds (from 0 to 1)
  - Precision is on the y-axis and recall is on the x-axis
- Area under the precision-recall curve serves as the guideline of how well the algorithm is doing
  - The bigger the area under the curve, the better
- This plot should be used when there is a moderate-to-high class imbalance

Precision-recall curve: the visual of tradeoff

- New version of `sckit-learn` now has a handy `plot_precision_recall_curve` function
- Just pass the model, the test data, and test labels to it

```
rf_prec_recall =
metrics.plot_precision_recall_curve(forest,
                              X_test, y_test,
                              name = "RF")
plt.show()
```



F1: precision vs recall

- $F1 = 2 \times \frac{(PR*RE)}{(PR+RE)}$
- A score that gives us a numeric value of the precision vs recall tradeoff
- The higher the $F1$ score, the better (the score can be a value between 0 and 1)

```
forest_f1 = metrics.f1_score(y_test,
                             forest_y_predict)
print(forest_f1)
```

```
0.9596730245231607
```

- In the fraud detection example our precision was 0, so was our recall, what would be the $F1$ score then?
- We want to use this score when we need to **maintain the balance between detecting as many** fraudulent transactions **as possible**, **while also making sure** that our detection is **precise**

Fbeta: precision vs recall (weighted)

- The main **drawback** of $F1$ is that it **assigns equal weights** to precision and recall
- Modified $F1$ where precision vs recall don't have to have the same weights is called F_β
    - Where β is the weight we can regulate to assign more weight to either precision or recall
- $F_\beta = (1+\beta^2) \times \frac{(PR*RE)}{(\beta^2*PR+RE)}$

```
forest_fbeta = metrics.fbeta_score(y_test,
                                   forest_y_predict,
                                   beta = 0.5)
print(forest_fbeta)
```

```
0.9506586050529044
```

- $F1$ and F_{beta} can also be used for multiclass classification problems
- If $\beta = 0.5$, what metric will be weighted more - precision or recall?
- Try out a few values of β and see how the score changes
- If we make $\beta = 1$ what will this score become?

Measuring performance: it’s a team effort

- In general using a single metric is often misleading and can lead to wrong, and sometimes disastrous conclusions
- Multiple metrics should be used to get the full picture of how well the model did
- If you were to choose a single metric though, which one would you pick? Why?

Log loss and why is it important

- Log (logarithmic) loss, a.k.a. **binary crossentropy** is a very interesting and important metric to know especially for those who would like to use and understand Neural Networks and their performance
- It is also widely used to **assess** conventional binary classification problems
- It can be extended to multi-class classification problems

Log loss: formula and intuition

- $L_{\log}(y,p) = -\log \Pr(y|p)$- the negative log likelihood of probability of y given p
  - Or the more frequently seen expanded version of it $L_{\log}(y,p) = -(y\log(p) + (1-y)\log(1-p))$
  - The p stands for probability of predicting 1 (class that is labeled $y=1$)
- A metric that takes into account the *uncertainty of your prediction based on how much it varies from the actual label*
- It **uses prediction probabilities of a class**, not the actual assigned class label
- Log loss is minimized when we want the algorithm to perform, so **the smaller this number, the better** (it can be 0 or greater)
- Read more in `scikit-learn` user guide ***here***

Log loss using scikit-learn metrics

- As it goes, scikit-learn has `log_loss` function to give us the log loss score
- All we need to do is to provide actual class labels and predicted probability scores as arguments

```
# The second argument is an array of predicted probabilities, not labels!
forest_log_loss = metrics.log_loss(y_test, forest_y_predict_prob[:, 1], eps=1e-15)
print ("Log loss: ", forest_log_loss)
```

```
Log loss:  0.21947942349408847
```

- Log loss here is small, but we don't really have a scale against which to measure it, do we?

Log loss using scikit-learn metrics - cont'd

- To have a more intuitive score that would convert it to a `0` to `1` scale, we could use exp as the inverse function of log
  - Since L_{log} is a negative log, we need reverse its sign before passing this to exp function

```
# Convert a difficult to interpret log loss to an overall accuracy in predicted probabilities.
print("Overall accuracy: ", math.exp(-forest_log_loss))
```

```
Overall accuracy:  0.8029366791543784
```

Loss curve: ideal case

- Loss curve can be constructed to view how the value changes based on different probability thresholds (similar to our precision vs recall plot)

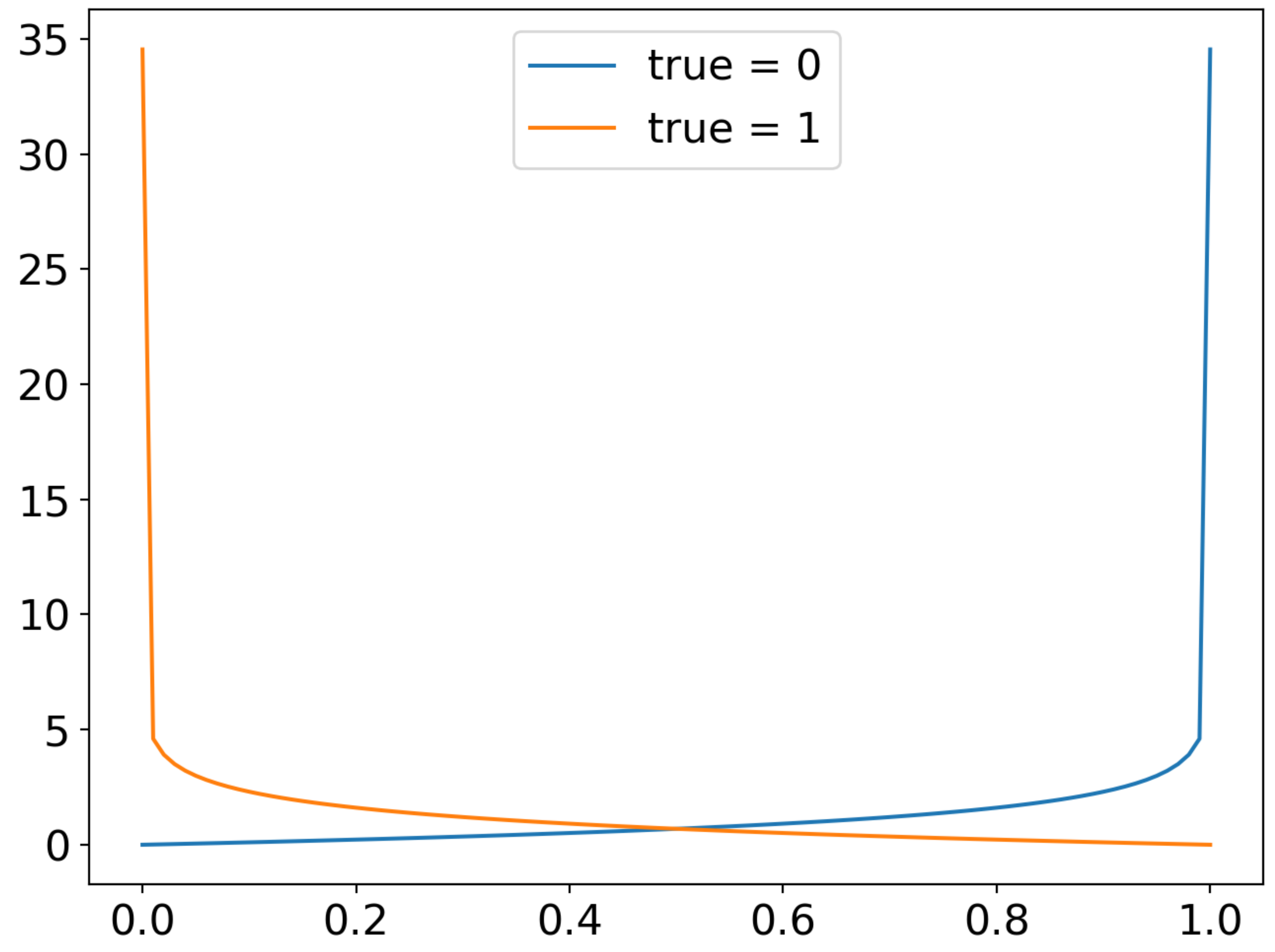
```
# Probability values: 0 to 1 in 0.01 increments.
prob_increments = [x*0.01 for x in range(0, 101)]

# Evaluate predictions for when true value is 0.
loss_0 = [metrics.log_loss([0], [x], labels=[0,1]) for x in prob_increments]

# Evaluate predictions for when true value is 1.
loss_1 = [metrics.log_loss([1], [x], labels=[0,1]) for x in prob_increments]
```

- Notice that we are going to construct it for both cases
    - When $y = 1$ being a `TRUE` prediction and $y = 0$ being a `TRUE` prediction

Loss curve: ideal case (cont'd)

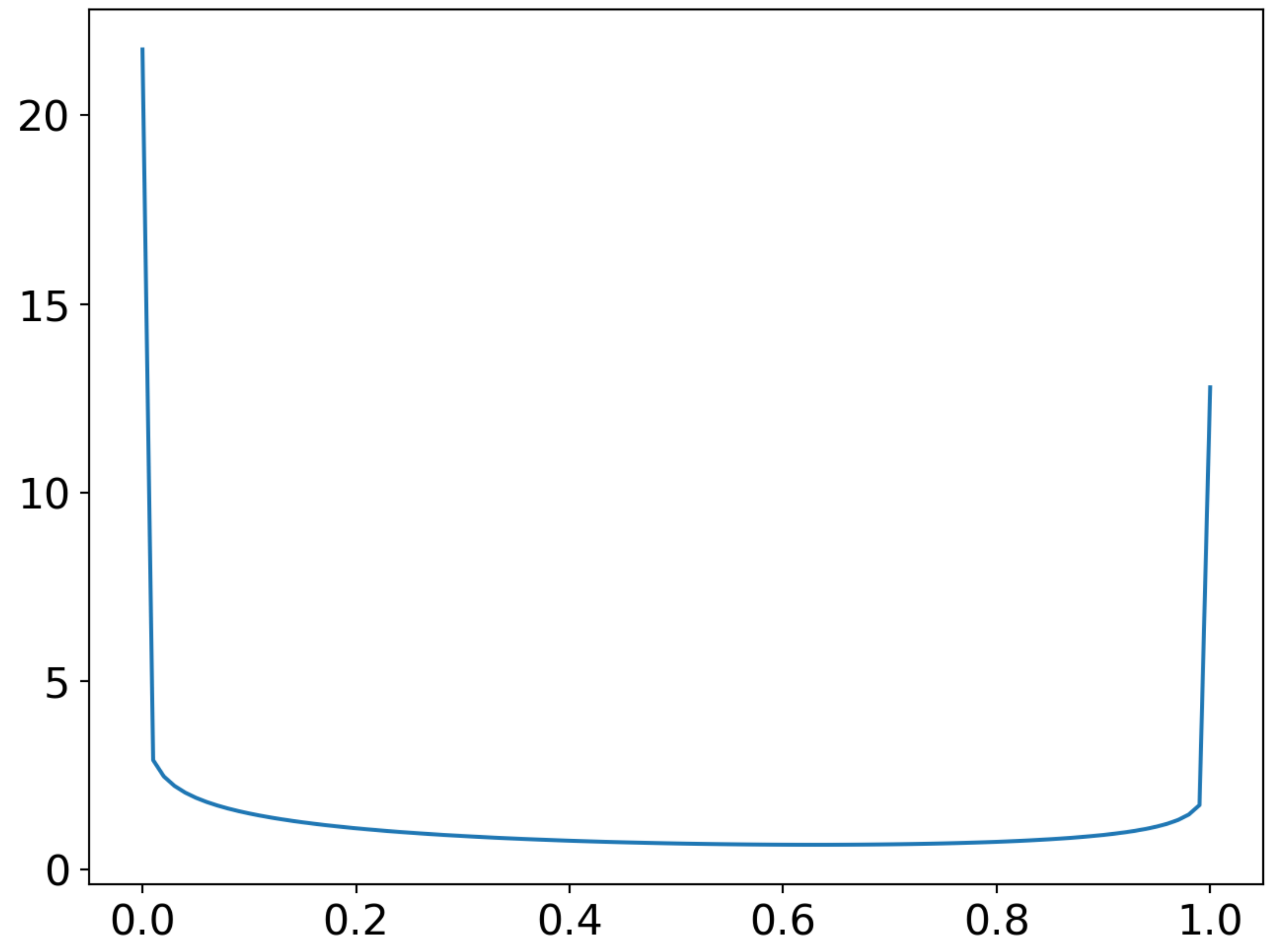
```
# Plot probability increments vs loss curves.
plt.plot(prob_increments,
         loss_0,
         label='true = 0')
plt.plot(prob_increments,
         loss_1,
         label='true = 1')
plt.legend()
plt.show()
```

- In the ideal case the log loss for both scenarios (true label = `0` and true label = `1`) the curve is perfectly symmetric
- It is minimized in the middle
- The penalty (big values of loss) are high for both we incorrectly give high probability to class `1`, when the actual value is `0` and vice versa

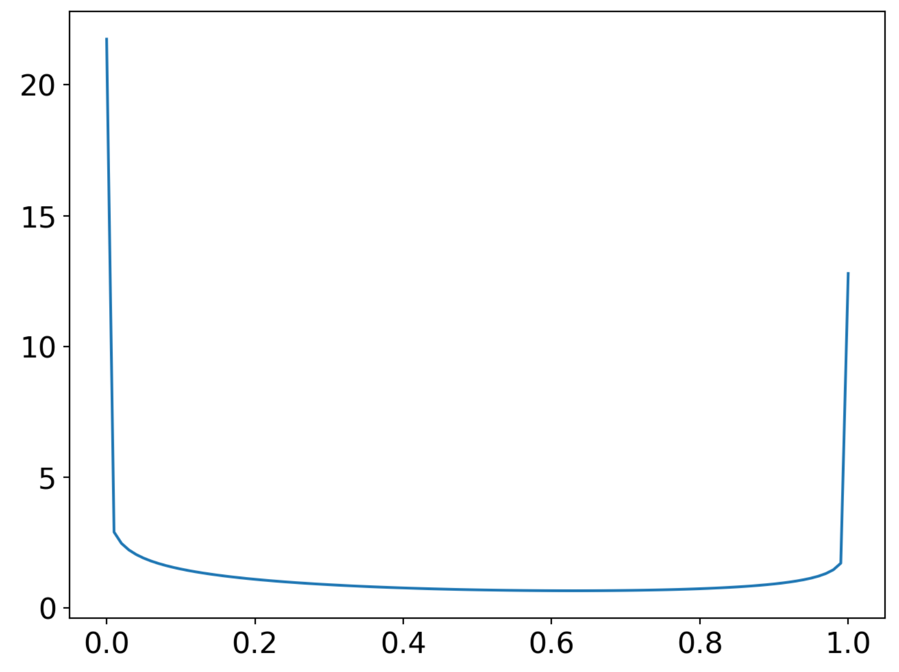


Loss curve: based on our data

```
# Loss for predicting different fixed probability
values.
losses = [metrics.log_loss(y_test, [y for x in
range(len(y_test))]) for y in prob_increments]
# Plot predictions vs loss.
plt.plot(prob_increments, losses)
plt.show()
```



Loss curve: based on our data (cont'd)



- The drawback of this evaluation metric can be seen when we have a class imbalance (in our case it's not big, but it is present)
- The penalty is bigger when we give higher probability to class `0` when in fact the class is `1`
- On the other hand the penalty is not so high for when we give higher probability to class `1`, when the true class is `0`
- Since **log loss score reports the average** of all of the values on this curve, we have no way of telling by just looking at the score without the plot if there might be an issue with it!

Receiver Operator Characteristic (ROC) curve

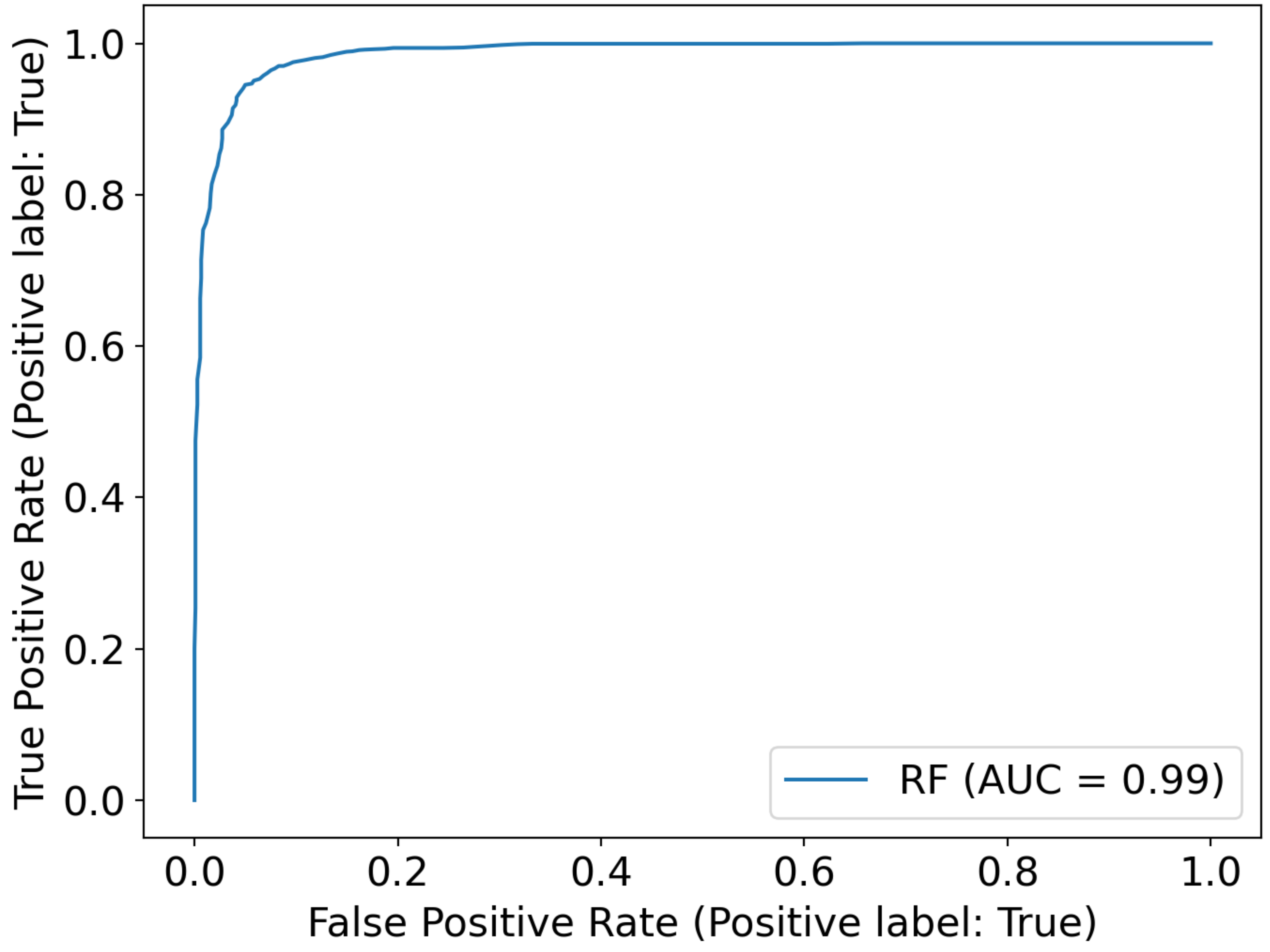
- The last (but not least) evaluation curve and metric that comes with it that we want to look at is the **tradeoff between True Positive Rate (a.k.a. Recall) and False Positive Rate (1-Specificity)**
- The curve is better known as **Receiver Operator Characteristic**
- As you may have already guessed, it is constructed based on possible probability thresholds (just like loss curve, and precision-recall curve)
- It is well suited for (fairly) balanced datasets
- The AUC (area under the ROC curve) is used to indicate how well the algorithm did
    - The closer to `1` the better, anything below `0.5` is considered worse performance than a random guess!

ROC curve: the tradeoff

- We can easily plot ROC curve using `plot_roc_curve` function, it takes the same arguments as the precision-recall plot we have made before

```
rf_roc = metrics.plot_roc_curve(forest,
                                X_test,
                                y_test,
                                name = "RF")
plt.show()
```

- As we have already seen, our RF model performed well on all fronts
- The AUC for this curve is about 99%, which is about as good as it ever gets in real life



AUC: area under the ROC curve

- To compute a single AUC score without the plot, we can use the `roc_auc_score` function

```
# Where y_pred are probabilities and y_true are binary class labels
forest_auc = metrics.roc_auc_score(y_test, forest_y_predict_prob[:, 1])
print("AUC: ", forest_auc)
```

```
AUC:  0.986855647527701
```

Wrapping score evaluation into function

```python
def get_performance_scores(y_test, y_predict, y_predict_prob, eps=1e-15, beta=0.5):
    from sklearn import metrics
    # Scores keys.
    metric_keys = ["accuracy", "precision", "recall", "f1", "fbeta", "log_loss", "AUC"]
    # Score values.
    metric_values = [None]*len(metric_keys)
    metric_values[0] = metrics.accuracy_score(y_test, y_predict)
    metric_values[1] = metrics.precision_score(y_test, y_predict)
    metric_values[2] = metrics.recall_score(y_test, y_predict)
    metric_values[3] = metrics.f1_score(y_test, y_predict)
    metric_values[4] = metrics.fbeta_score(y_test, y_predict, beta=beta)
    metric_values[5] = metrics.log_loss(y_test, y_predict_prob[:, 1], eps=eps)
    metric_values[6] = metrics.roc_auc_score(y_test, y_predict_prob[:, 1])
    perf_metrics = dict(zip(metric_keys, metric_values))
    return(perf_metrics)
```

Test score generating function

```
forest_scores = get_performance_scores(y_test, forest_y_predict, forest_y_predict_prob)

metrics_forest = {"RF": forest_scores}
print(metrics_forest)
```

```
{'RF': {'accuracy': 0.9483960948396095, 'precision': 0.9447424892703863, 'recall':
0.9750830564784053, 'f1': 0.9596730245231607, 'fbeta': 0.9506586050529044, 'log_loss':
0.21947942349408847, 'AUC': 0.986855647527701}}
```

Knowledge check 1

Exercise 1

Module completion checklist

Objective	Complete
Explain metrics used to assess binary classifier performance	✔
Evaluate base random forest model	✔
Optimize random forest model using RandomizedCV method	
Use performance metrics to compare optimized RF to base RF model	

Vanilla RF model parameters

- Let’s take a look at the model parameters in the default RF model

```
forest.get_params()
```

```
{'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini',
'max_depth': None, 'max_features': 'auto', 'max_leaf_nodes': None, 'max_samples': 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': 100, 'n_jobs':
None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
```

- Given our previous evaluation metrics, the default model performed very well
- Can we do even better?
- The only way to find out is to try different combinations of parameters and see which one yields the best results

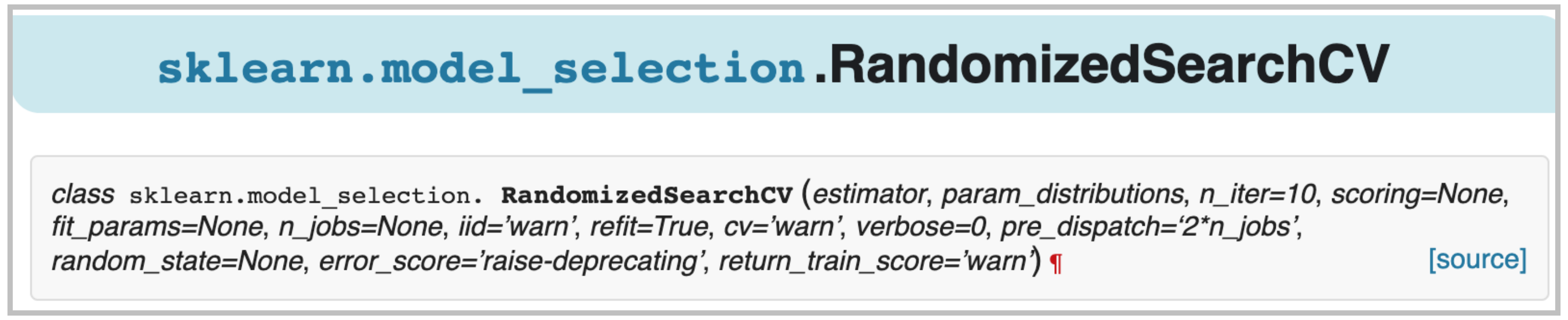
Note: sometimes the cost of parameter evaluation exceeds the benefit, so it’s not always the best approach, but for smaller and tractable datasets, it’s often the way to go!

Tuning Random Forests model

- Common methods to try out combinations of different parameter values:
  - `GridSearchCV`: trying out **all possible combinations**; can be computationally expensive, even when dealing with a handful of parameters and different values for each
  - `RandomizedSearchCV`: trying out a **random sample of combinations** of different parameters; a less compute intensive alternative to `GridSearchCV`

RandomizedSearchCV

- It uses a randomized search on hyperparameters
- Unlike `GridSearchCV`, it samples only a fixed number of parameter settings from the pool of all possible combinations of parameters and values



sklearn.model_selection.RandomizedSearchCV

class sklearn.model_selection. **RandomizedSearchCV** (*estimator, param_distributions, n_iter=10, scoring=None, fit_params=None, n_jobs=None, iid='warn', refit=True, cv='warn', verbose=0, pre_dispatch='2*n_jobs', random_state=None, error_score='raise-deprecating', return_train_score='warn'*) ¶ [source]

- To read more about this function and the arguments it takes, visit the official ***documentation page***

Parameter grid

- Let’s create a grid of parameter ranges

```python
# Number of trees in random forest.
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 20)]

# Number of features to consider at every split.
max_features = ['auto', 'sqrt']

# Maximum number of levels in tree.
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)

# Minimum number of samples required to split a node.
min_samples_split = [2, 5, 10]

# Minimum number of samples required at each leaf node.
min_samples_leaf = [1, 2, 4]

# Set Minimal Cost-Complexity Pruning parameter (has to be >= 0.0).
ccp_alpha = [0.0, 0.001, 0.01, 0.1, 0.2, 0.3]
```

Set up RandomizedSearchCV function

- Now we instantiate the model, using 3-fold cross-validation with 100 different parameter settings that are sampled

```
rf_random = RandomizedSearchCV(estimator = forest, #<- model object
                               param_distributions = random_grid, #<- param grid
                               n_iter = 100,#<- number of param. settings sampled
                               cv = 3,      #<- 3-fold CV
                               verbose = 0, #<- silence lengthy output to console
                               random_state = 1, #<- set random state
                               n_jobs = -1)      #<- use all available processors
# Fit the random search model.
rf_random.fit(X_train, y_train) #<- fit like any other scikit-learn model
```

```
# Take a look at optimal combination of parameters.
print(rf_random.best_params_)
```

```
{'n_estimators': 768, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_features': 'auto',
'max_depth': 110, 'ccp_alpha': 0.0}
```

Module completion checklist

Objective	Complete
Explain metrics used to assess binary classifier performance	✔
Evaluate base random forest model	✔
Optimize random forest model using RandomizedCV method	✔
Use performance metrics to compare optimized RF to base RF model	

Optimized RF model

- Now we can use these optimized hyperparameters to implement the random forest again on `X_train`
- You can pass the parameters directly from the result of our randomized search
    - In Python you would use `**parameter_dictionary` to pass a dictionary with named parameter values in the form `'parameter_name': parameter_value`

```
# Pass best parameters obtained through randomized search to RF classifier.
optimized_forest = RandomForestClassifier(**rf_random.best_params_)

# Train the optimized RF model.
optimized_forest.fit(X_train, y_train)
```

```
RandomForestClassifier(max_depth=110, n_estimators=768)
```

Predict and compute performance scores

- Get predictions for test dataset using optimized RF model

```
# Get predicted labels for test data.
optimized_forest_y_predict = optimized_forest.predict(X_test)

# Get predicted probabilities.
optimized_forest_y_predict_proba = optimized_forest.predict_proba(X_test)
```

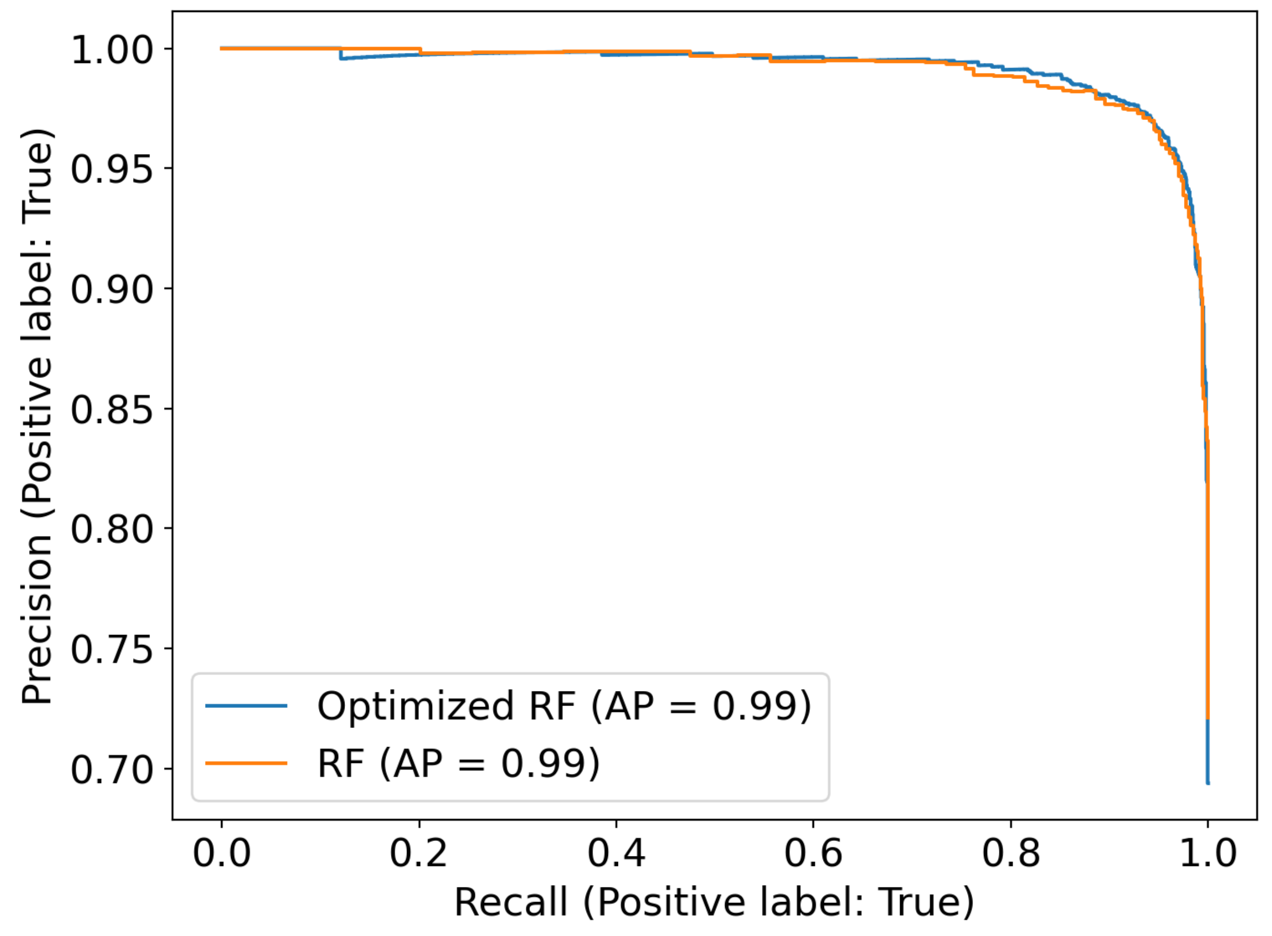
```
# Compute performance scores.
optimized_forest_scores = get_performance_scores(y_test,
                                                  optimized_forest_y_predict,
                                                  optimized_forest_y_predict_proba)
```

- Let's leave the discussion of individual metrics for each model for the end, when we collect multiple results
- Instead, we will just look at the precision-vs-recall and ROC plots for models compared to each other

Precision vs recall curve: the tradeoff

- Plot precision vs recall curve for the optimized model
- Add RF curve the previously plotted to it

```
ax = plt.gca() #<- create a new axis object
opt_rf_prec_recall =
metrics.plot_precision_recall_curve(optimized_forest,
                                    X_test,
                                    y_test,
                                    ax = ax,
                                    name = "Optimized RF")
rf_prec_recall.plot(ax = ax, name = "RF") #<- add rf plot
plt.show()
```

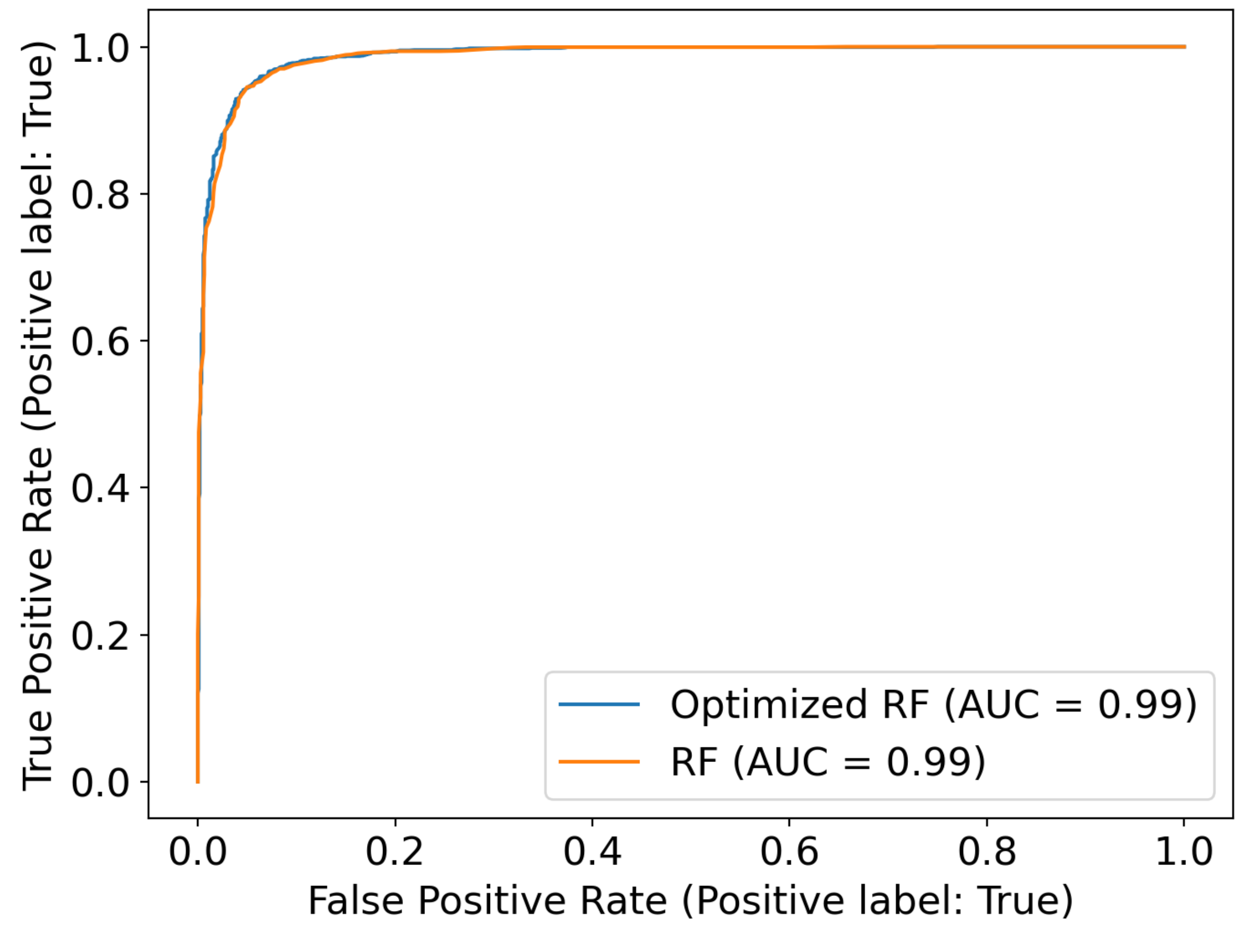


- We can see from this plot that both RF and optimized RF were very close to each other (almost identical)

ROC curve: the tradeoff

- Plot ROC curve for the optimized model
- Add RF curve the previously plotted to it

```
ax = plt.gca()
opt_rf_roc = metrics.plot_roc_curve(optimized_forest,
                       X_test,
                       y_test,
                       name = "Optimized RF",
                       ax = ax)

rf_roc.plot(ax = ax, name = "RF")
plt.show()
```



- Same goes for the ROC plot
- This makes us wonder whether the computationally expensive randomized search was worth the effort/resources?
- What are you thoughts on that?

Optimized RF: append to metrics dictionary

- We will need all scores for all models in one place in order for us to plot them against each other
- Let's append the optimized RF scores to the master dictionary

```
metrics_forest.update({"Optimized RF": optimized_forest_scores})
print(metrics_forest)
```

```
{'RF': {'accuracy': 0.9483960948396095, 'precision': 0.9447424892703863, 'recall':
0.9750830564784053, 'f1': 0.9596730245231607, 'fbeta': 0.9506586050529044, 'log_loss':
0.21947942349408847, 'AUC': 0.986855647527701}, 'Optimized RF': {'accuracy':
0.9483960948396095, 'precision': 0.9400212314225053, 'recall': 0.9806201550387597, 'f1':
0.9598915989159891, 'fbeta': 0.9478698351530723, 'log_loss': 0.21667604220201855, 'AUC':
0.9876901226920936}}
```

Knowledge check 2

Exercise 2

Module completion checklist

Objective	Complete
Explain metrics used to assess binary classifier performance	✔
Evaluate base random forest model	✔
Optimize random forest model using RandomizedCV method	✔
Use performance metrics to compare optimized RF to base RF model	✔

Summary

- Today we:
  - Introduced gradient boosting and how it compares to bagging
  - Implemented various classification model performance metrics and discuss their interpretation
  - Optimized a random forest model to predict vulnerable households
  - Learned about optimizing gradient boosting models and compare their performance
- In the next class we will:
  - Implement base GBM model and asses its performance
  - Optimize GBM model using RandomizedCV method
  - Use performance metrics to compare all ensemble methods
  - Introduce the concept of a hyperplane and classification using a hyperplane
  - Summarize the idea of maximal margin classifier and its pitfalls

Congratulations on completing this module!