



Advanced Classification - Part 6

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

Module completion checklist

Objective	Complete
Summarize the key differences between support vector classifier and support vector machine	
Build a support vector machine model to classify the Costa Rica dataset	
Optimize the support vector machine model using grid search	

Loading packages

- Let's load the packages we will be using
- These packages are used for classification in SVM

```
import os
import pickle
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pathlib import Path
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn import metrics
from sklearn.model_selection import GridSearchCV
```

Goal

- We already used our Costa Rican dataset and classified the poverty levels of the individuals
- We are going to do the same today
- We will determine whether a person is classified as **poor or not** based on the information we have about that person's living conditions and educational qualifications
- We will build 2 models today
 - Support vector machine, and
 - Optimized support vector machine using grid search

Classification with non-linear boundary

- What if our data cannot be classified using linear boundary?
- SVCs can be useless in a highly non-linear class boundary
- In that case, we can introduce a **classifier with a non-linear decision boundary** or non-linear hyperplane
- Support vector classifiers with a non-linear decision boundary are called **support vector machines (SVM)**

Transforming the features

- The idea is to transform the p features into a higher dimensional space
- For example, if we are transforming in quadratic space, we convert p features to a $2p$ feature dimension

Consider p features

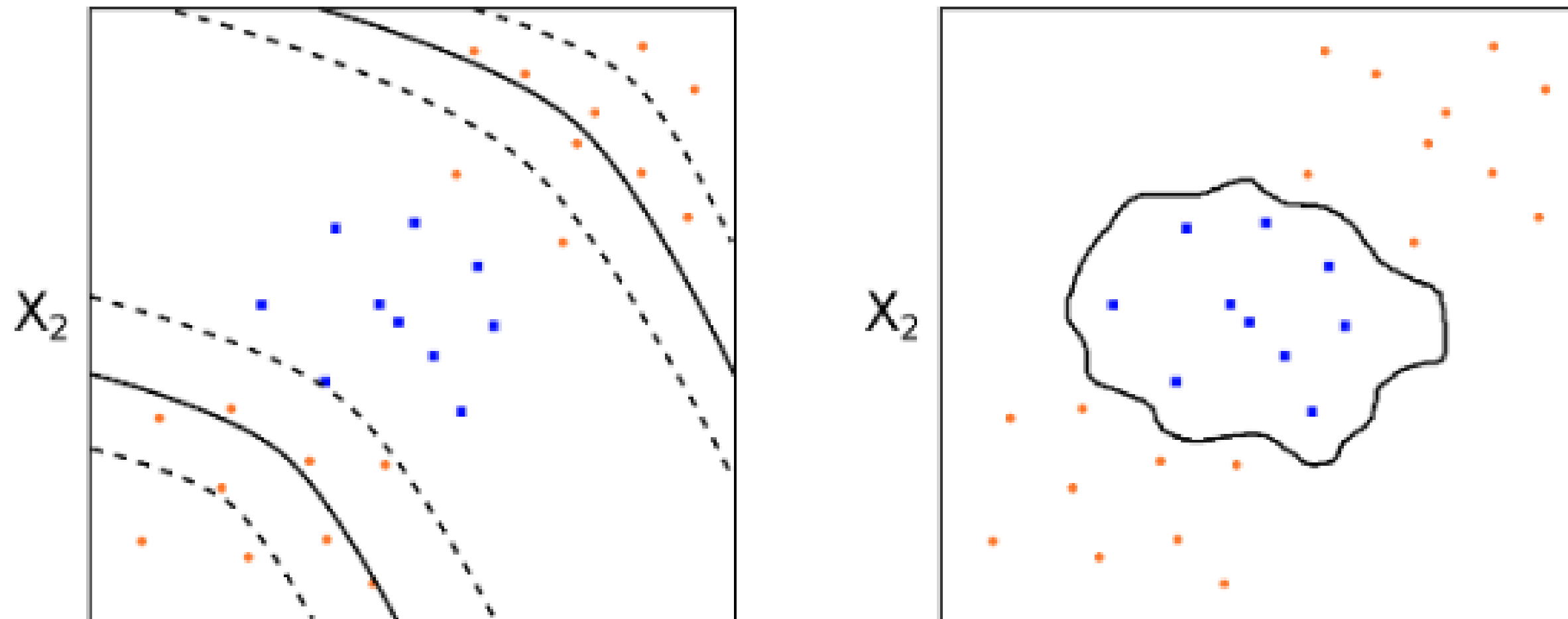
$$x_1, x_2, \dots, x_n$$

They are transformed to $2p$ features

$$x_1, x_1^2, \dots, x_n, x_n^2$$

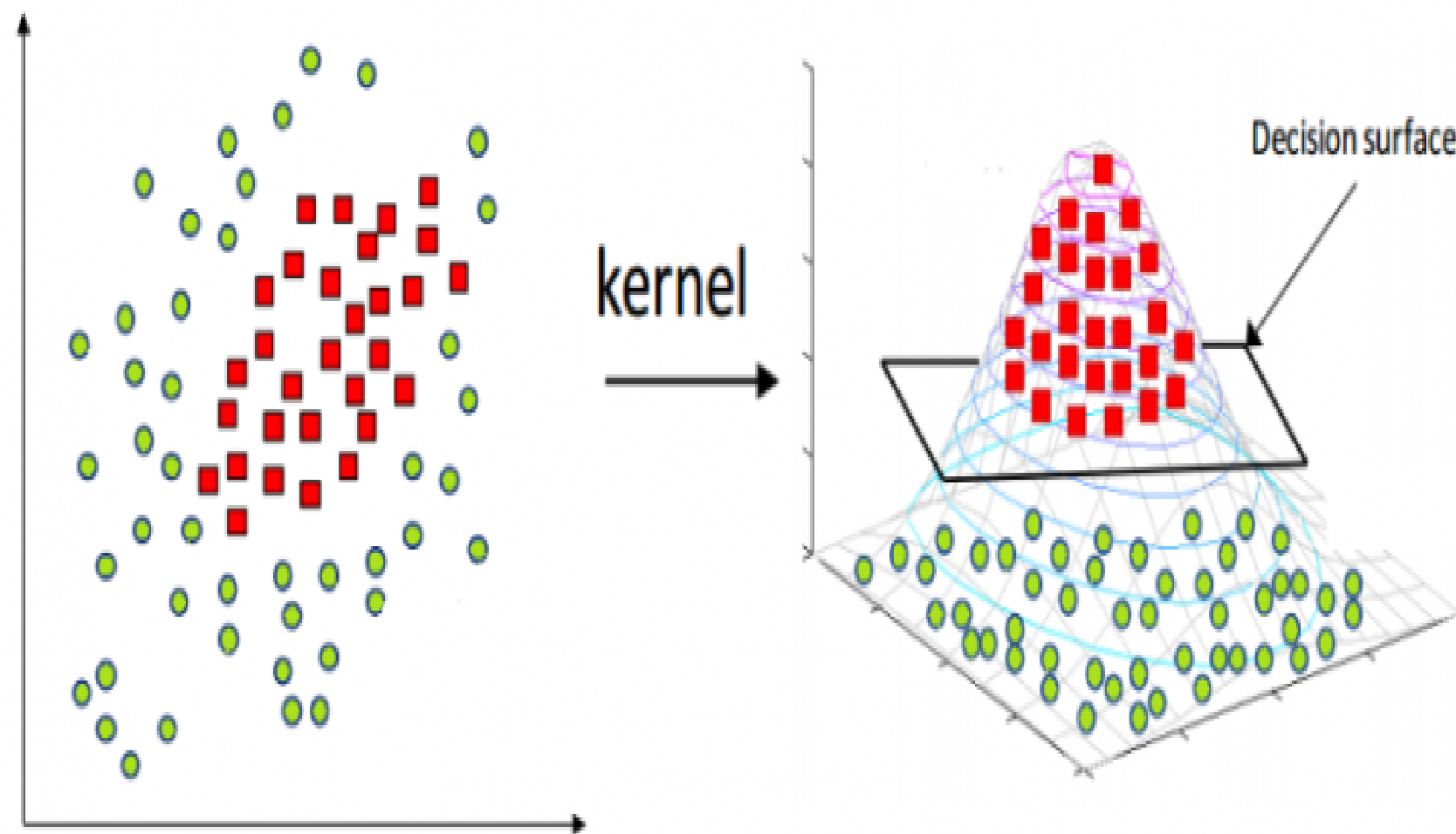
Support vector machine

- The decision boundary is non-linear in p -dimensional space, but it is linear in the new transformed dimensional space
- This is called the kernel approach and the classifier is called the **support vector machines (SVM)**
- It can use **quadratic, cubic, or even higher order polynomial functions**
- The problem with it is that we start to accumulate more features since they are transformed and computation becomes unmanageable quickly



Decision boundary

- The decision boundary is non-linear in p -dimensional space, but it is linear in the new transformed dimensional space
- We have different kernels (non linear boundary conditions) to do it
 - radial
 - polynomial
 - quadratic
 - cubic



Kernel: radial basis function

- Today we will use radial basis function (RBF) kernel for our data
- RBF kernel is the general purpose kernel used when there is no prior knowledge about the data
- Consider there are two observations \mathbf{x} and \mathbf{x}' - the RBF kernel is defined by the mathematical equation

$$K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2)$$

Kernel: gamma value

- Gamma is the value which controls the shape of the kernel
- The default value of gamma is `scale` which means it uses $1 / (n_features * \text{variance of } X)$
- If `auto`, uses $1 / n_features$
- Alternatively one can pass float values to gamma
- We will use an arbitrary value of `0.011` for gamma

Module completion checklist

Objective	Complete
Summarize the key differences between support vector classifier and support vector machine	✓
Build a support vector machine model to classify the Costa Rica dataset	
Optimize the support vector machine model using grid search	

Review data cleaning steps

- Today, we will be loading the cleaned dataset we used earlier
- 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
 - Remove highly correlated variables

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 pickled dataset, `costa_clean`
- Save it as `costa_clean`

```
costa_clean = pickle.load(open((data_dir + "/costa_clean.sav"), "rb"))
```

```
# Print the head.  
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 on data

- Let's view the column names

```
# Print the columns.  
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', 'cookenrgy_elec',  
      'cookenrgy_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_buena_vista', 'urban_rural', 'rural_type', 'local', 'Tercet_L1']
```

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 with 70% train and 30% test.  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)
```


Build a SVM model

```
# Build the SVM model.  
# Note here that the kernel rbf means radial kernel.  
sv_machine = SVC(kernel = 'rbf',  
                  gamma = 0.011,  
                  probability = True)  
sv_machine.fit(X_train, y_train)
```

```
SVC(gamma=0.011, probability=True)
```

Predict on the test dataset

```
# Predict on the test dataset.
svm_y_predict = sv_machine.predict(X_test)
svm_y_predict[0:5]

# Predict on test, but instead of labels
# we will get probabilities for class 0 and 1.
```

```
array([ True,  True,  True,  True,  True])
```

```
svm_y_predict_prob = sv_machine.predict_proba(X_test)
print(svm_y_predict_prob[5:])
```

```
[[0.40850243 0.59149757]
 [0.10654403 0.89345597]
 [0.1395987  0.8604013 ]
 ...
 [0.13136736 0.86863264]
 [0.83573684 0.16426316]
 [0.86367169 0.13632831]]
```

Recap: score evaluation function

- Earlier, we used a function to wrap all performance score calculations into one block of code which can be called repeatedly
- We will use this function to evaluate our SVM models today

```
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)
```

Predict on the test dataset

```
svm_scores = get_performance_scores(y_test, svm_y_predict, svm_y_predict_prob)
print(svm_scores)
```

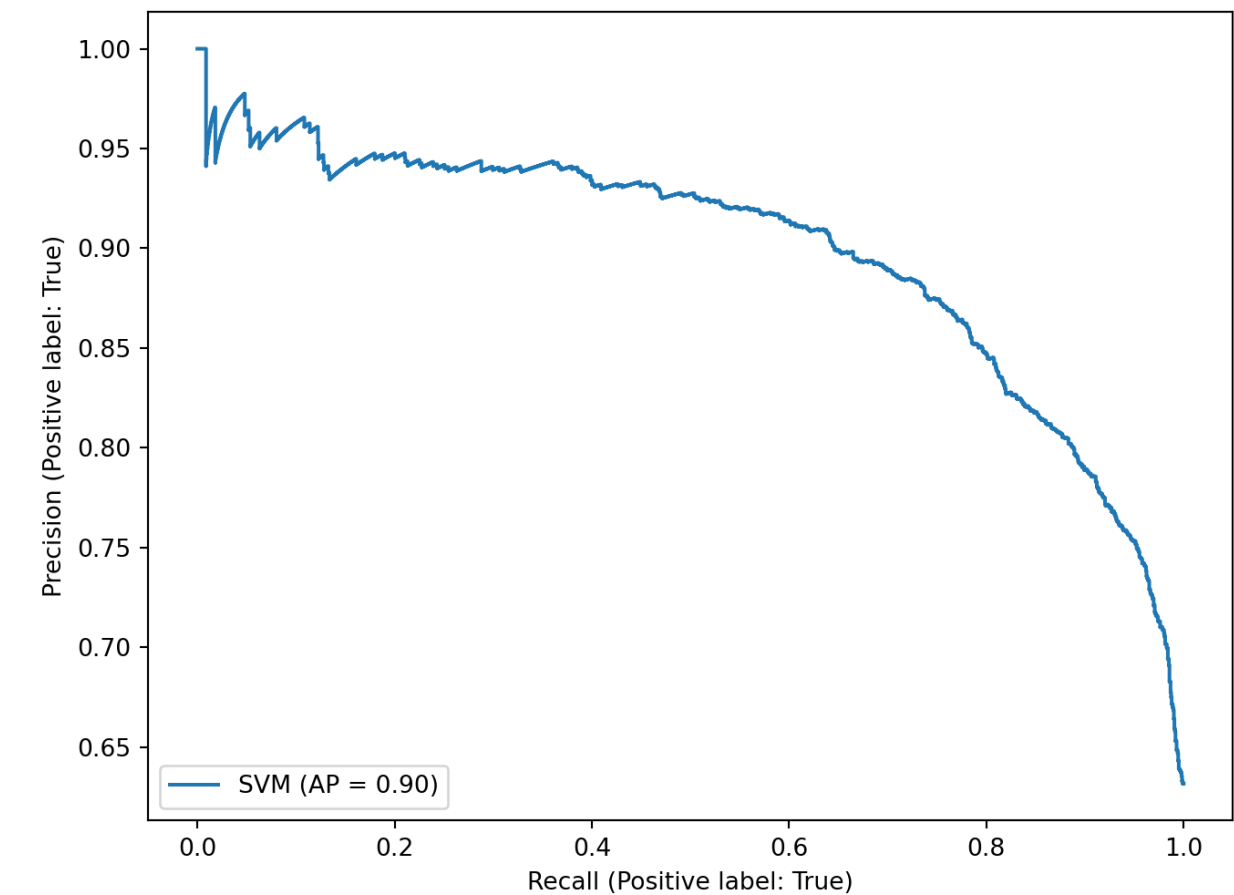
```
{'accuracy': 0.7904463040446305, 'precision': 0.8032209360845496, 'recall':  
0.8837209302325582, 'f1': 0.8415502240970207, 'fbeta': 0.8181258970678695, 'log_loss':  
0.4574972925256918, 'AUC': 0.855928553701514}
```

Precision vs recall curve: the tradeoff

- Plot precision vs recall curve for the SVM model

```
ax = plt.gca() #<- create a new axis object
svm_prec_recall =
metrics.plot_precision_recall_curve(sv_machine,
                                   X_test,
                                   y_test,
                                   ax = ax,
                                   name = "SVM")

plt.show()
```

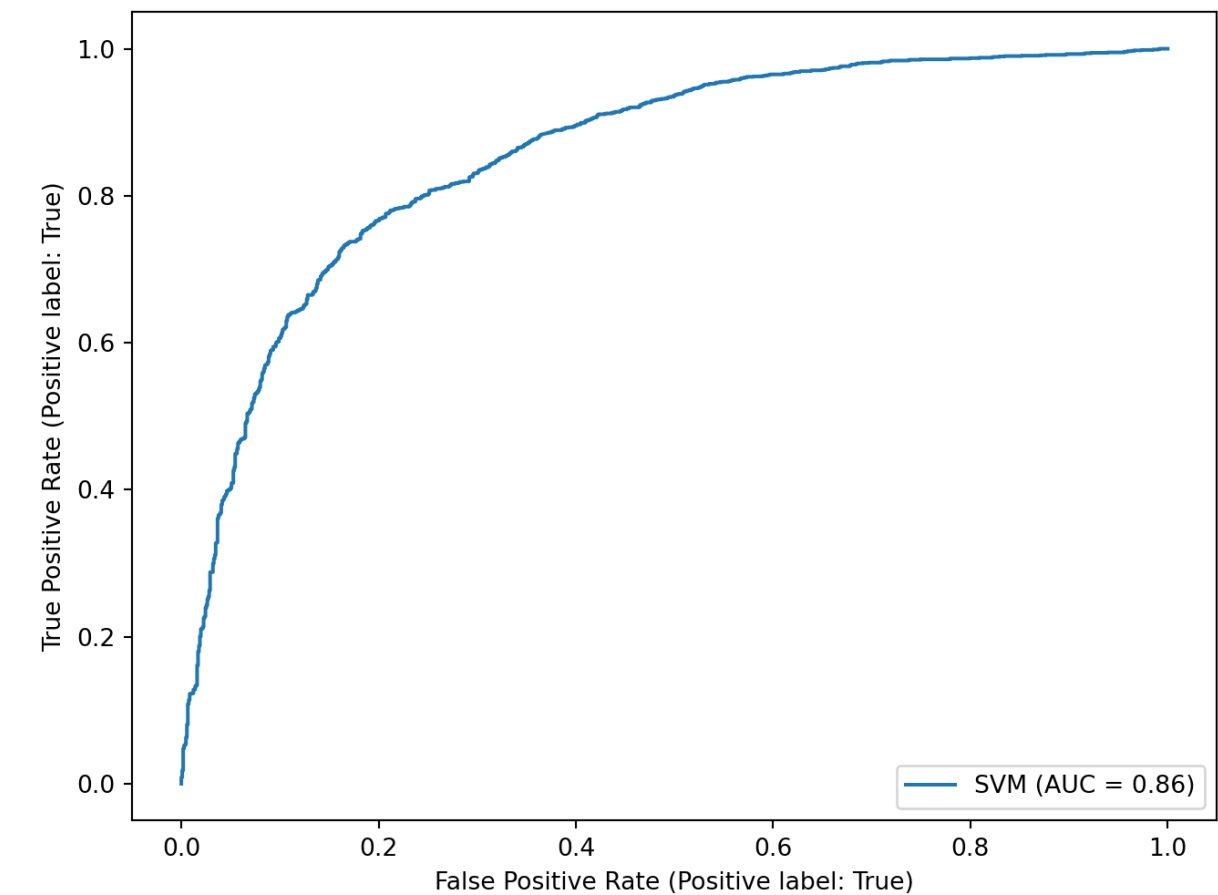


ROC curve: the tradeoff

- Plot ROC curve for the SVM model

```
ax = plt.gca()
svm_roc = metrics.plot_roc_curve(sv_machine,
                                X_test,
                                y_test,
                                name = "SVM",
                                ax = ax)

plt.show()
```



Save final metrics of SVM

- Let's save our svm score in our `metrics_svm` dataset
- We first have to load our `metrics_svm` dataframe which we created previously

```
metrics_svm = pickle.load(open((data_dir +  
"/metrics_svm.sav"), "rb"))
```

Save final metrics of SVM

```
# Add the model to our dataframe.  
metrics_svm.update({"SVM": svm_scores})  
print(metrics_svm)
```

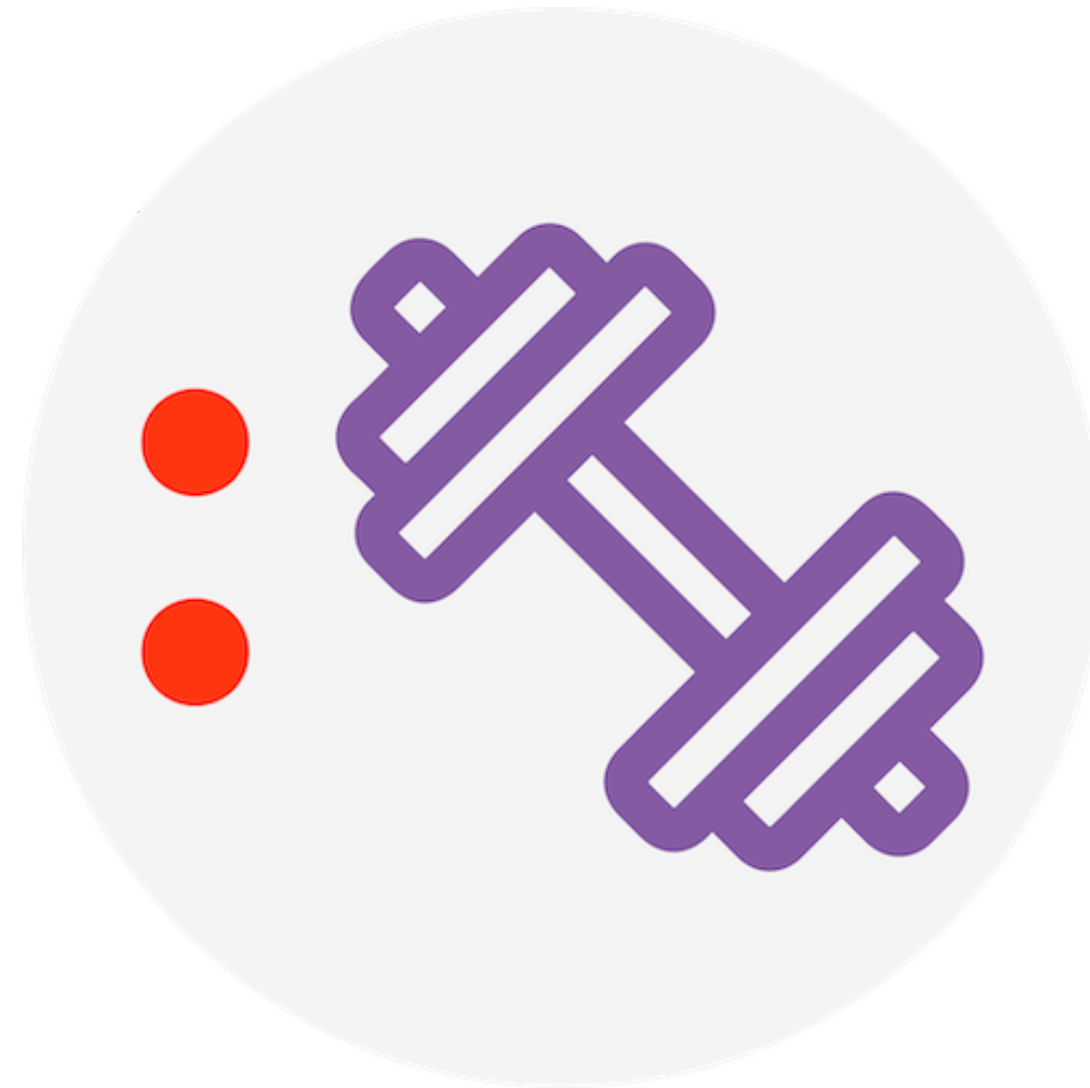
```
{ 'RF': { 'accuracy': 0.9483960948396095, 'precision': 0.9447424892703863, 'recall':  
0.9750830564784053, 'f1': 0.9596730245231607, 'fbeta': 0.9506586050529044, 'log_loss':  
0.21947942349408847, 'AUC': 0.986855647527701}, 'Optimized RF': { 'accuracy':  
0.9483960948396095, 'precision': 0.9400212314225053, 'recall': 0.9806201550387597, 'f1':  
0.9598915989159891, 'fbeta': 0.9478698351530723, 'log_loss': 0.21667604220201855, 'AUC':  
0.9876901226920936}, 'GBM': { 'accuracy': 0.8291492329149233, 'precision':  
0.8374358974358974, 'recall': 0.9042081949058693, 'f1': 0.869542066027689, 'fbeta':  
0.8499895898396836, 'log_loss': 0.39102395649143923, 'AUC': 0.902052011186816}, 'Optimized  
GBM': { 'accuracy': 0.9605997210599722, 'precision': 0.9684560044272275, 'recall':  
0.9689922480620154, 'f1': 0.9687240520343205, 'fbeta': 0.9685632056674784, 'log_loss':  
0.5487445996283076, 'AUC': 0.9918971705530634}, 'SVC': { 'accuracy': 0.7890516039051604,  
'precision': 0.8081067213955875, 'recall': 0.872093023255814, 'f1': 0.8388814913448736,  
'fbeta': 0.8201416371589252, 'log_loss': 0.45830503084971225, 'AUC': 0.85334770267762},  
'SVM': { 'accuracy': 0.7904463040446305, 'precision': 0.8032209360845496, 'recall':  
0.8837209302325582, 'f1': 0.8415502240970207, 'fbeta': 0.8181258970678695, 'log_loss':  
0.4574972925256918, 'AUC': 0.855928553701514}}
```

- The performance of SVM increased slightly when compared to SVC mostly because our decision boundary is non-linear

Knowledge check 1



Exercise 1



Module completion checklist

Objective	Complete
Summarize the key differences between support vector classifier and support vector machine	✓
Build a support vector machine model to classify the Costa Rica dataset	✓
Optimize the support vector machine model using grid search	

Optimize the SVM model with grid search

- Until now, we have used arbitrary values for C and γ
- **Grid search** gives a range of n values to a particular parameter in the model and lets the model choose the **best tuning parameter from the best model**
- Here, we give the range of values to both C and γ parameters to choose the best tuning parameter and do 5 fold cross-validation

Grid search and cv on SVM model

```
# Set the parameters by cross-validation.
tuned_parameters = [{'kernel': ['rbf'], 'gamma': [1e-3, 1e-4],
                                     'C': [1, 10, 100, 1000]}]

# Fit the tuned parameters for grid search with 5 fold cross-validation to SVM.
svm_cv = GridSearchCV(SVC(), tuned_parameters, cv = 5)
svm_cv.fit(X_train, y_train)
```

```
GridSearchCV(cv=5, estimator=SVC(),
             param_grid=[{'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
                           'kernel': ['rbf']}])
```

Find best parameters

```
# Find the best tuned parameters.  
print(svm_cv.best_params_)
```

```
{'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}
```

```
# Extract the best hyperparameters.  
optimized_c = svm_cv.best_params_['C']  
optimized_gamma = svm_cv.best_params_['gamma']  
optimized_kernel = svm_cv.best_params_['kernel']
```

- Thus, we find that the optimum C value was very different from our default value of 1

Fit the best parameters to build the optimized model

```
# Run the model with optimized hyperparameters.  
sv_cv_optimized = SVC(kernel = optimized_kernel,  
                      gamma = optimized_gamma,  
                      C = optimized_c,  
                      probability = True)  
  
sv_cv_optimized.fit(X_train, y_train)
```

```
SVC(C=1000, gamma=0.001, probability=True)
```

Predict on the test dataset

```
# Predict on the test dataset.
opt_svm_y_predict = sv_cv_optimized.predict(X_test)
opt_svm_y_predict[0:5]

# Predict on test, but instead of labels
# we will get probabilities for class 0 and 1.
```

```
array([ True,  True,  True,  True,  True])
```

```
opt_svm_y_predict_prob = sv_cv_optimized.predict_proba(X_test)
print(opt_svm_y_predict_prob[5:])
```

```
[[0.75667416 0.24332584]
 [0.00745365 0.99254635]
 [0.01396175 0.98603825]
 ...
 [0.12930068 0.87069932]
 [0.51275975 0.48724025]
 [0.85417574 0.14582426]]
```


Predict on the test dataset

```
opt_svm_scores = get_performance_scores(y_test, opt_svm_y_predict, opt_svm_y_predict_prob)
print(opt_svm_scores)
```

```
{'accuracy': 0.8315899581589958, 'precision': 0.8573743922204214, 'recall':  
0.8787375415282392, 'f1': 0.8679245283018868, 'fbeta': 0.8615635179153094, 'log_loss':  
0.3939702321910344, 'AUC': 0.8962732511214971}
```

- As we can see, our optimized SVM model performed much better than the original SVM model

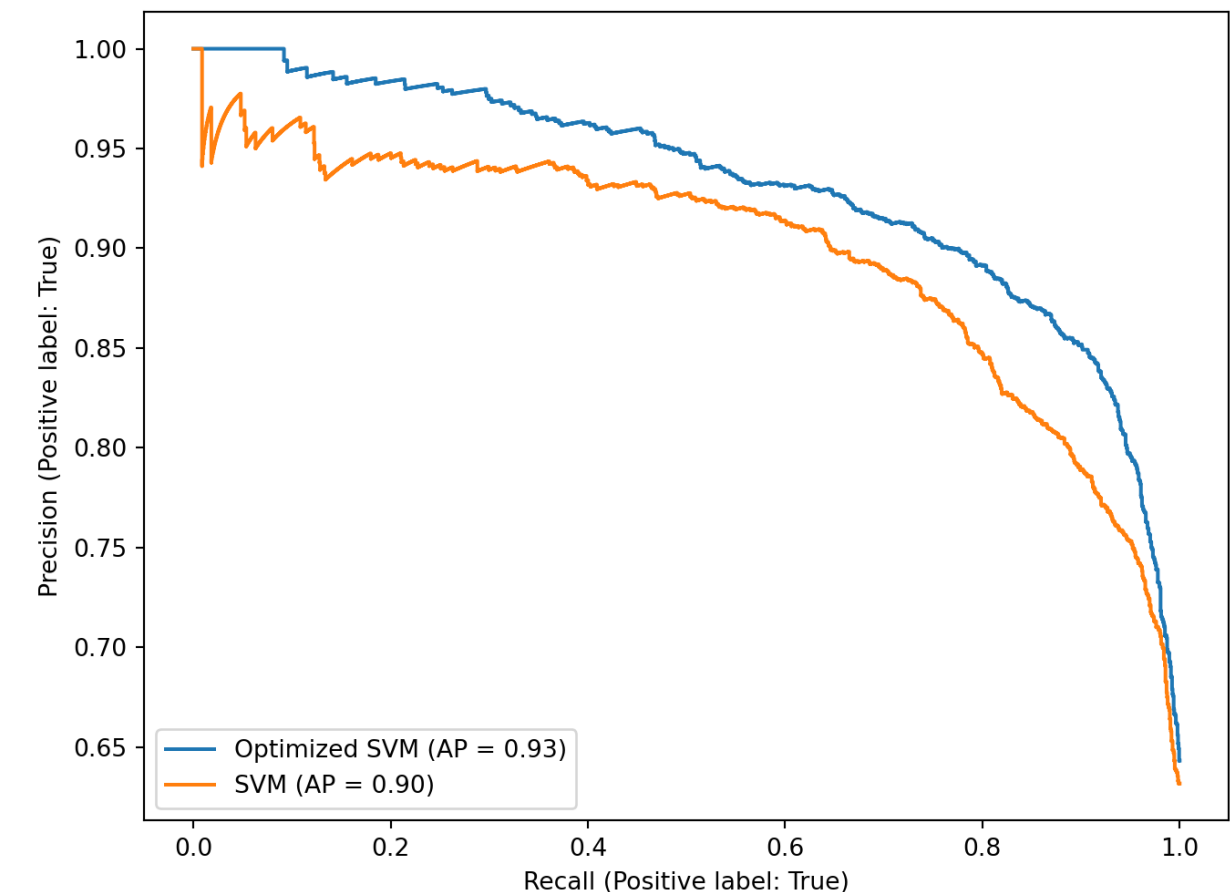
Precision vs recall curve: the tradeoff

- Plot precision vs recall curve for the SVC model
- Add SVM curve we previously plotted to it

```
ax = plt.gca() #<- create a new axis object
opt_svm_prec_recall =
metrics.plot_precision_recall_curve(sv_cv_optimized,
                                   X_test,
                                   y_test,
                                   ax = ax,
                                   name = "Optimized SVM")

svm_prec_recall.plot(ax = ax, name = "SVM") #<- add rf plot
plt.show()
```

- As we can see, our optimized SVM model performed much better than the original SVM model and the SVC model

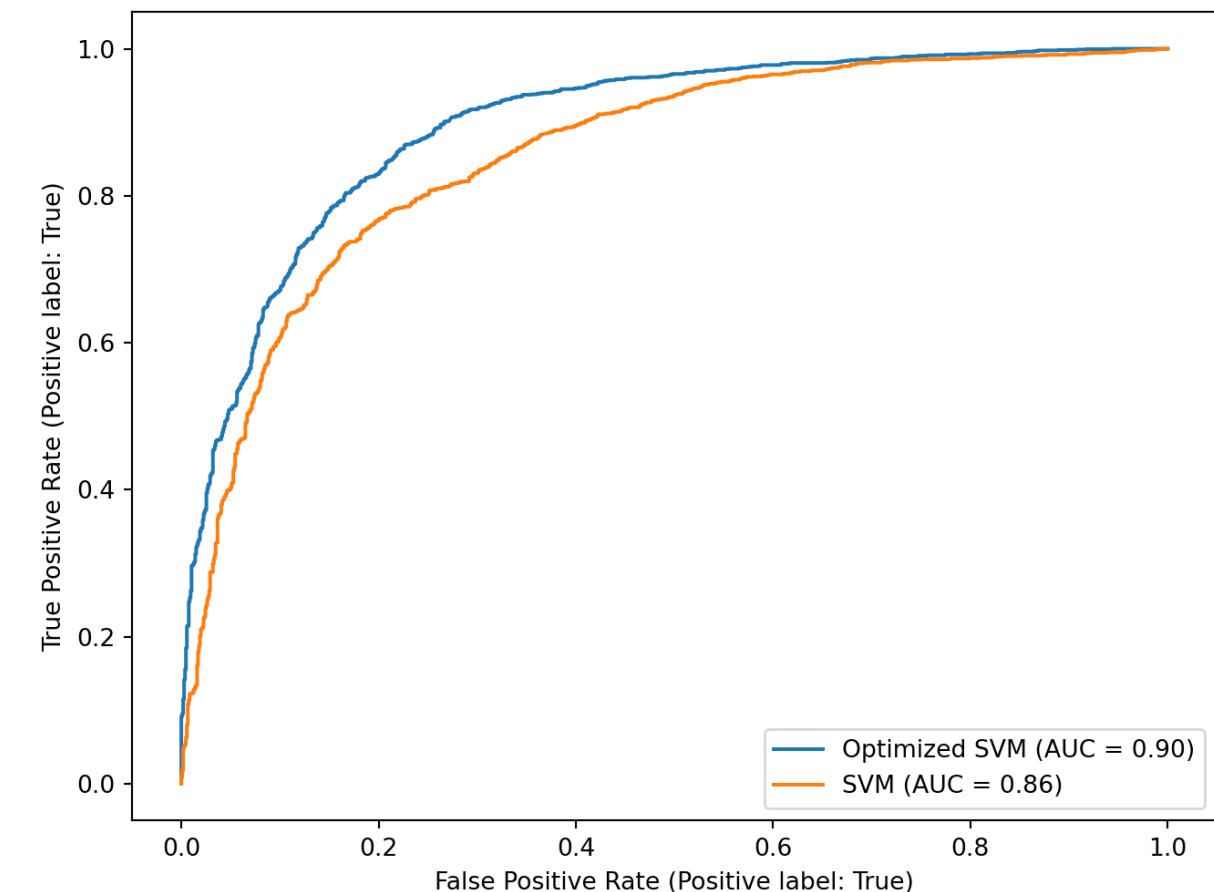


ROC curve: the tradeoff

- Plot ROC curve for the SVM model
- Add SVC and SVM model we previously plotted to it

```
ax = plt.gca()
opt_svm_roc = metrics.plot_roc_curve(sv_cv_optimized,
                                     X_test,
                                     y_test,
                                     name = "Optimized SVM",
                                     ax = ax)

svm_roc.plot(ax = ax, name = "SVM")
plt.show()
```



- The AUC for this curve is about 90%, which is good!
- As we can see, our optimized SVM model has a higher AUC than the original SVM model

Save final metrics of SVM

```
# Add the model to our dataframe.  
metrics_svm.update({"Optimized SVM": opt_svm_scores})  
print(metrics_svm)
```

```
{'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}, 'GBM': {'accuracy': 0.8291492329149233, 'precision':  
0.8374358974358974, 'recall': 0.9042081949058693, 'f1': 0.869542066027689, 'fbeta':  
0.8499895898396836, 'log_loss': 0.39102395649143923, 'AUC': 0.902052011186816}, 'Optimized  
GBM': {'accuracy': 0.9605997210599722, 'precision': 0.9684560044272275, 'recall':  
0.9689922480620154, 'f1': 0.9687240520343205, 'fbeta': 0.9685632056674784, 'log_loss':  
0.5487445996283076, 'AUC': 0.9918971705530634}, 'SVC': {'accuracy': 0.7890516039051604,  
'precision': 0.8081067213955875, 'recall': 0.872093023255814, 'f1': 0.8388814913448736,  
'fbeta': 0.8201416371589252, 'log_loss': 0.45830503084971225, 'AUC': 0.85334770267762},  
'SVM': {'accuracy': 0.7904463040446305, 'precision': 0.8032209360845496, 'recall':  
0.8837209302325582, 'f1': 0.8415502240970207, 'fbeta': 0.8181258970678695, 'log_loss':  
0.4574972925256918, 'AUC': 0.855928553701514}, 'Optimized SVM': {'accuracy':  
0.8315899581589958, 'precision': 0.8573743922204214, 'recall': 0.8787375415282392, 'f1':  
0.8679245283018868, 'fbeta': 0.8615635179153094, 'log_loss': 0.3939702321910344, 'AUC':  
0.8862722511214071}}
```

- The accuracy of our optimized SVM has increased here since we found the optimal tuning parameters and ran it with a cross-validation

Create comparison plot using function

- Use the previously created `compare_metrics` function to create the comparison plot

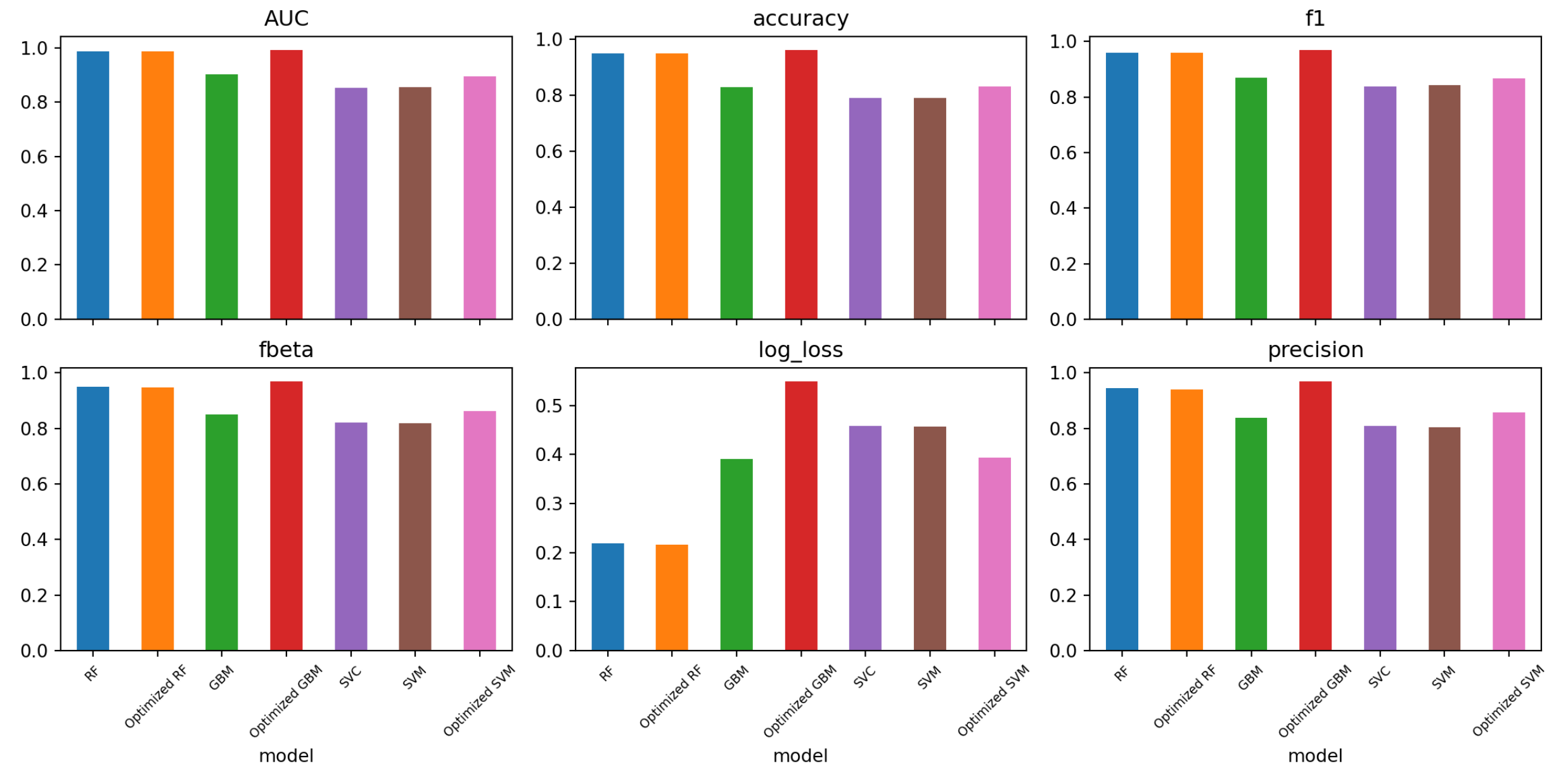
```
def compare_metrics(metrics_dict, color_list = None):
    metrics_df = pd.DataFrame(metrics_dict)
    metrics_df["metric"] = metrics_df.index
    metrics_df = metrics_df.reset_index(drop = True)
    metrics_long = pd.melt(metrics_df, id_vars = "metric", var_name = "model",
                           value_vars = list(metrics_dict.keys()))

    if color_list is None:
        cmap = plt.rcParams['axes.prop_cycle'].by_key()['color']
        colors = cmap[:len(metrics_dict.keys())]
    else:
        colors = color_list

    fig, axes = plt.subplots(2, 3, figsize = (12, 6))
    for (metric, group), ax in zip(metrics_long.groupby("metric"), axes.flatten()):
        group.plot(x = 'model', y = 'value', kind = 'bar', color = colors, ax = ax,
                   title = metric, legend = None, sharex = True)
        ax.xaxis.set_tick_params(rotation = 45, labelsiz=7)
    plt.tight_layout(0.5)
    return((fig, axes))
```

Create comparison plot using function

```
fig, axes =  
compare_metrics(metrics_svm)  
plt.show()
```



SVM performance

- Although ensemble classifiers outperformed SVM, it would still probably perform better than most of the single classifier models
- This is because we have a lot of categorical variables and SVM models them better than the other models
- We also noted that our model performance improved after optimization
- We could increase our grid search range to improve the performance further, however this would come at the cost of computational complexity

Advantages and disadvantages of SVM

- **Advantages**

- It is an effective classifier for **high dimensional data**
- **Memory efficient**, since only the support vectors are needed to be stored in memory for making the classification decision
- Versatile by allowing **different boundary conditions**

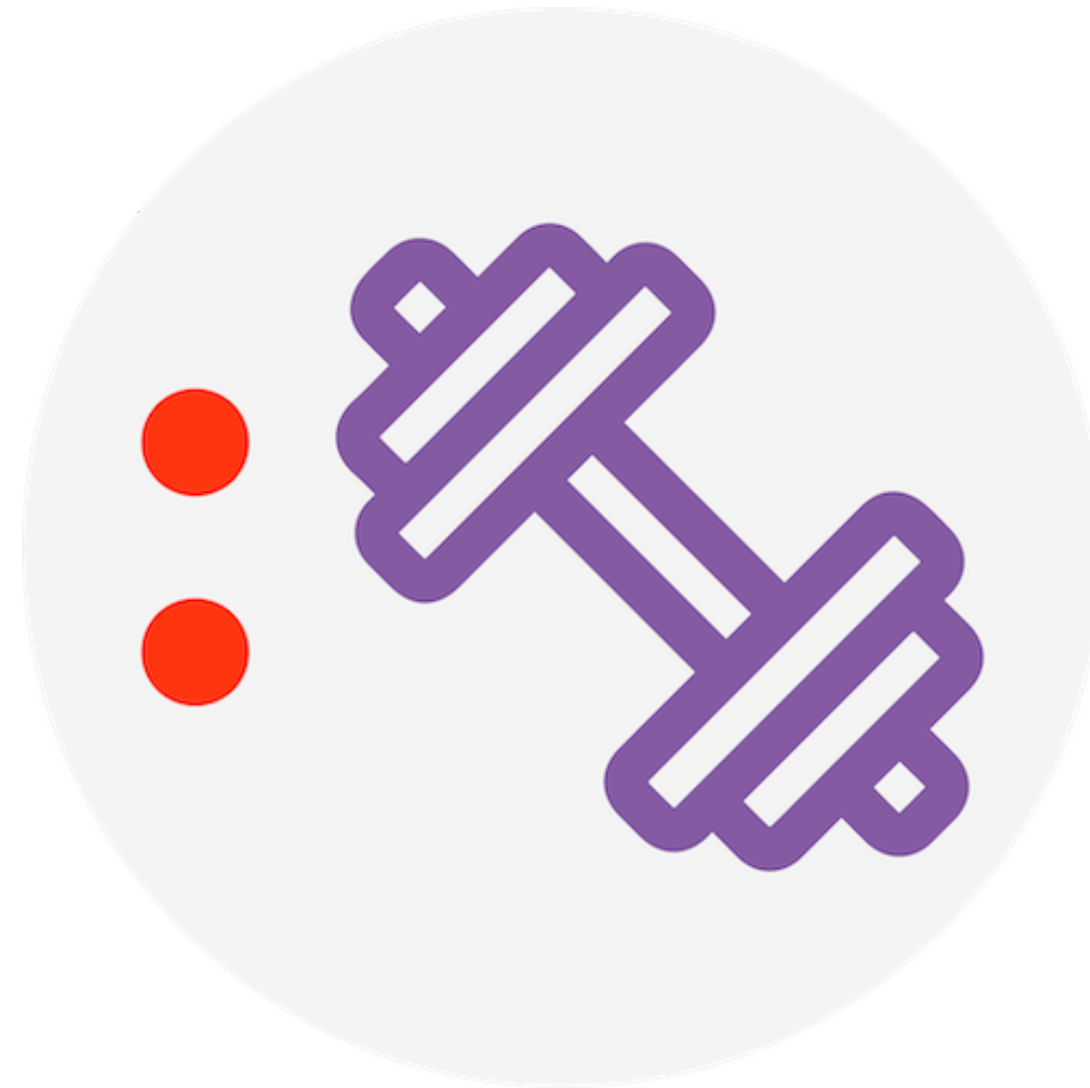
- **Disadvantages**

- It is **non-probabilistic**, so there is no probabilistic interpretation for group membership
- When $p > n$, i.e. when number of predictors exceeds the number of observations, SVM performs poorly

Knowledge check 2



Exercise 2



Module completion checklist

Objective	Complete
Summarize the key differences between support vector classifier and support vector machine	✓
Build a support vector machine model to classify the Costa Rica dataset	✓
Optimize the support vector machine model using grid search	✓

Summary

- Today we learned:
 - concepts of hyperplanes and Support Vector Machines
 - how SVMs can be used to create non-linear classifications
- There are many data science areas you can explore next like recommender systems, neural networks, deep learning, text mining, and others!

Congratulations on completing this module!

