lab_jupyter_svm

May 15, 2022

1 Support Vector Machine

Estimated time needed: **30** minutes

In this lab, you will learn and obtain hands-on practices on Support Vector Machine model.

We will be using a real-world diabetes food items suggestion dataset, which contains detailed nutrition information about a food item. The objective is to classify what food a diabetic patient should choose More Often or Less Often for a specific food item given its nutrients.

1.1 Objectives

After completing this lab you will be able to:

- Train and evaluate SVM classifiers
- Tune important SVM hyperparameters such as regularization and kernel types
- Plot hyperplanes and margins from trained SVM models

1.2 SVM Overview

SVM tries to find hyperplanes that have the maximum margin. The hyperplanes are determined by support vectors (data points have the smallest distance to the hyperplanes). Meanwhile, in order to reduce model variance, the SVM model aims to find the maximum possible margins so that unseen data will be more likely to be classified correctly.

SVM addresses non-linear separable via kernel trick. Kernels are a special type of function that takes two vectors and returns a real number, like a dot-product operation. As such, kernels are not any real mapping functions from low dimensional spaces to high dimensional spaces.

For example, suppose we have two vectors $x = (x\_1, x\_2)$ and $y = (y\_1, y\_2)$

Now we have a simple polynomial kernel like the following:

$$k(x, y) = (x^T y)^2$$

If we apply the kernel on vector `x` and `y`, we will get:

$$k(x, y) = (x^T y)^2 = (x\_1y\_1 + x\_2y\_2)^2 = x\_1^2y\_1^2 + x\_2^2y\_2^2 + 2x\_1x\_2y\_1y\_2$$

It can be seen as a dot-product between two higher-dimensional vectors (`3-dimensional`):

$$\hat{x} = (x\_1^2, x\_2^2, \sqrt{2}x\_1x\_2)$$

$$\hat{y} = (y\_1^2, x\_2^2, \sqrt{2}y\_1y\_2)$$

As such, computing the `k(x, y)` is equivalent to computing a dot-product of the higher dimensional vectors, without doing the actual feature space transforms. Consequently, SVM with non-linear kernels can transform existing features into high dimensional features that can be linearly separated in higher dimensional spaces.

1.3 Setup lab environment

[1]:
```
# All Libraries required for this lab are listed below. The libraries␣
 ↪pre-installed on Skills Network Labs are commented.
# !mamba install -qy pandas==1.3.3 numpy==1.21.2 ipywidgets==7.4.2 scipy==7.4.2␣
 ↪tqdm==4.62.3 matplotlib==3.5.0 seaborn==0.9.0

# install imbalanced-learn package
!pip install imbalanced-learn==0.8.0
# Note: If your environment doesn't support "!mamba install", use "!pip␣
 ↪install"
```

```
Collecting imbalanced-learn==0.8.0
  Downloading imbalanced_learn-0.8.0-py3-none-any.whl (206 kB)
                             206.5/206.5 KB
22.2 MB/s eta 0:00:00
Requirement already satisfied: scipy>=0.19.1 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from imbalanced-
learn==0.8.0) (1.7.3)
Collecting joblib>=0.11
  Downloading joblib-1.1.0-py2.py3-none-any.whl (306 kB)
                             307.0/307.0 KB
25.5 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.13.3 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from imbalanced-
learn==0.8.0) (1.21.6)
Collecting scikit-learn>=0.24
  Downloading
scikit_learn-1.0.2-cp37-cp37m-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
(24.8 MB)
                              24.8/24.8 MB
50.7 MB/s eta 0:00:0000:0100:01
Collecting threadpoolctl>=2.0.0
  Downloading threadpoolctl-3.1.0-py3-none-any.whl (14 kB)
Installing collected packages: threadpoolctl, joblib, scikit-learn, imbalanced-
```

```
learn
  Attempting uninstall: scikit-learn
    Found existing installation: scikit-learn 0.20.1
    Uninstalling scikit-learn-0.20.1:
      Successfully uninstalled scikit-learn-0.20.1
Successfully installed imbalanced-learn-0.8.0 joblib-1.1.0 scikit-learn-1.0.2
threadpoolctl-3.1.0
```

[2]:
```
# Import required packages
import pandas as pd
import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import MinMaxScaler
# Evaluation metrics related methods
from sklearn.metrics import classification_report, accuracy_score, f1_score,␣
 ↪confusion_matrix, precision_recall_fscore_support, precision_score,␣
 ↪recall_score

import matplotlib.pyplot as plt
import seaborn as sns
from imblearn.under_sampling import RandomUnderSampler
%matplotlib inline
```

[3]:
```
# Setup a random seed to be 123
rs = 123
```

1.4 Load and explore dataset

Let's first load the dataset as a **Pandas** dataframe and conduct some basic explorations

[4]:
```
# Load the dataset
dataset_url = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.
 ↪cloud/IBM-ML241EN-SkillsNetwork/labs/datasets/food_items_binary.csv"
food_df = pd.read_csv(dataset_url)
```

and let's quickly looks at its first 5 rows

[5]:
```
food_df.head(10)
```

[5]:

	Calories	Total Fat	Saturated Fat	Monounsaturated Fat	\
0	149.0	0	0.0	0.0	
1	123.0	0	0.0	0.0	
2	150.0	0	0.0	0.0	
3	110.0	0	0.0	0.0	
4	143.0	0	0.0	0.0	

```
5     110.0          0            0.0                  0.0
6     142.0          0            0.0                  0.0
7     102.0          0            0.0                  0.0
8     145.0          0            0.0                  0.0
9     171.0          0            0.0                  0.0

   Polyunsaturated Fat  Trans Fat  Cholesterol  Sodium  Total Carbohydrate  \
0                  0.0        0.0            0     9.0                 9.8
1                  0.0        0.0            0     5.0                 6.6
2                  0.0        0.0            0     4.0                11.4
3                  0.0        0.0            0     6.0                 7.0
4                  0.0        0.0            0     7.0                13.1
5                  0.0        0.0            0     6.0                 7.0
6                  0.0        0.0            0    12.0                10.6
7                  0.0        0.0            0    13.0                 5.0
8                  0.0        0.0            0    17.0                11.0
9                  0.0        0.0            0     8.0                13.7

   Dietary Fiber  Sugars  Sugar Alcohol  Protein  Vitamin A  Vitamin C  \
0            0.0     0.0              0      1.3          0          0
1            0.0     0.0              0      0.8          0          0
2            0.0     0.0              0      1.3          0          0
3            0.0     0.0              0      0.8          0          0
4            0.0     0.0              0      1.0          0          0
5            0.0     0.0              0      0.8          0          0
6            0.0     0.0              0      1.2          0          0
7            0.0     0.0              0      0.7          0          0
8            0.0     0.0              0      1.2          0          0
9            0.0     0.0              0      2.5          0          0

   Calcium  Iron  class
0        0     0      0
1        0     0      0
2        0     0      0
3        0     0      0
4        0     0      0
5        0     0      0
6        0     0      0
7        0     0      0
8        0     0      0
9        0     0      0
```

[6]:
```
# Get the row entries with col 0 to -1 (16)
feature_cols = list(food_df.iloc[:, :-1].columns)
feature_cols
```

[6]:
```
['Calories',
 'Total Fat',
 'Saturated Fat',
 'Monounsaturated Fat',
 'Polyunsaturated Fat',
 'Trans Fat',
 'Cholesterol',
 'Sodium',
 'Total Carbohydrate',
 'Dietary Fiber',
 'Sugars',
 'Sugar Alcohol',
 'Protein',
 'Vitamin A',
 'Vitamin C',
 'Calcium',
 'Iron']
```

[7]:
```
X = food_df.iloc[:, :-1]
y = food_df.iloc[:, -1:]
```

[8]:
```
X.describe()
```

[8]:

	Calories	Total Fat	Saturated Fat	Monounsaturated Fat	\
count	7639.000000	7639.000000	7639.000000	7639.000000	
mean	105.098835	2.318235	0.368920	0.309216	
std	77.224368	3.438941	0.737164	1.310260	
min	0.000000	0.000000	0.000000	0.000000	
25%	50.000000	0.000000	0.000000	0.000000	
50%	100.000000	1.000000	0.000000	0.000000	
75%	140.000000	3.000000	0.500000	0.000000	
max	2210.000000	24.000000	8.000000	16.800000	

	Polyunsaturated Fat	Trans Fat	Cholesterol	Sodium	\
count	7639.000000	7639.000000	7639.000000	7639.000000	
mean	0.264116	0.007069	4.308679	235.053659	
std	2.848250	0.094783	14.788162	252.438163	
min	0.000000	0.000000	0.000000	0.000000	
25%	0.000000	0.000000	0.000000	20.000000	
50%	0.000000	0.000000	0.000000	150.000000	
75%	0.000000	0.000000	0.000000	375.500000	
max	235.000000	2.500000	450.000000	2220.000000	

	Total Carbohydrate	Dietary Fiber	Sugars	Sugar Alcohol	\
count	7639.000000	7639.000000	7639.000000	7639.000000	
mean	15.510719	1.806074	3.734756	0.059039	
std	14.028570	4.099947	5.013685	0.771173	

```
min              0.000000       0.000000      0.000000       0.000000
25%              3.700000       0.000000      0.000000       0.000000
50%             13.000000       1.000000      2.000000       0.000000
75%             24.000000       3.000000      5.000000       0.000000
max            270.000000     305.000000     39.000000      19.000000
```

```
           Protein    Vitamin A    Vitamin C      Calcium         Iron
count  7639.000000  7639.000000  7639.000000  7639.000000  7639.000000
mean      4.298975     6.636733     6.487237     3.937688     5.510014
std       5.349881    19.658111    19.566500     7.892694     9.855960
min       0.000000     0.000000     0.000000     0.000000     0.000000
25%       0.100000     0.000000     0.000000     0.000000     0.000000
50%       3.000000     0.000000     0.000000     0.000000     2.000000
75%       6.000000     6.000000     4.000000     4.000000     8.000000
max      70.000000   370.000000   280.000000   110.000000   100.000000
```

as we can see from the outputs above, this food item dataset contains 17 types of nutrients about a food item such as Calories, Total Fat, Protein, Sugar, and so on, as numeric variables.

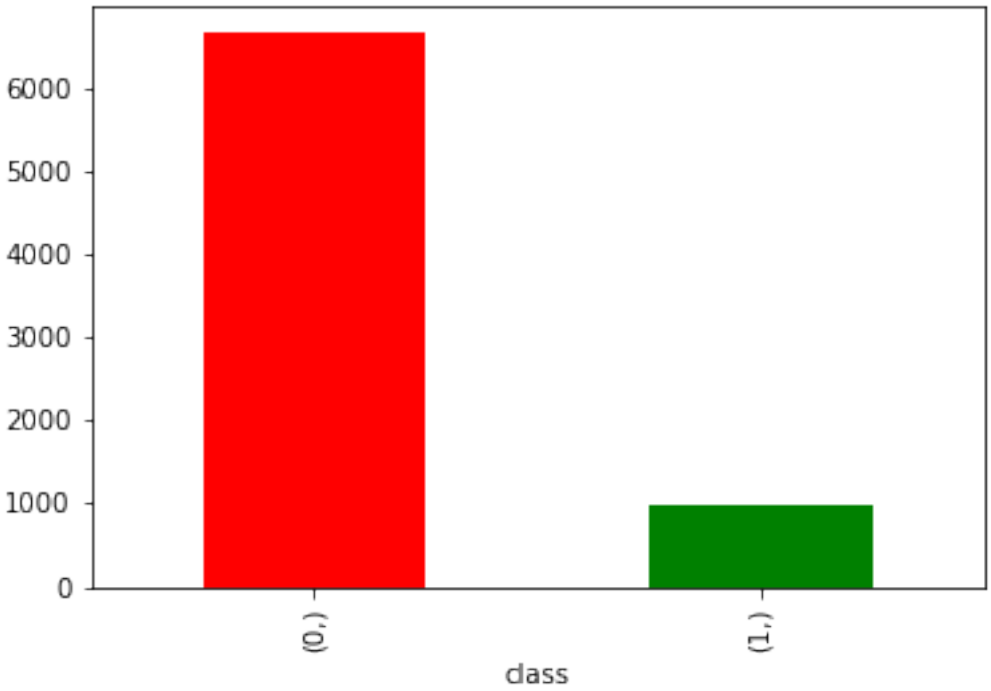
Next, let's check the target variable, such as the `class` column to see what are label values and their distribution.

[9]:
```
# # Get the row entries with the last col 'class'
y.value_counts(normalize=True)
```

[9]:
```
class
0        0.870402
1        0.129598
dtype: float64
```

[10]:
```
y.value_counts().plot.bar(color=['red', 'green'])
```

[10]:
```
<AxesSubplot:xlabel='class'>
```



As we can see from the bar chart above, this dataset has two classes `Less Often` and `More Often`. The two labels are imbalanced with most food items should be chosen less often for diabetic patients.

1.5 Build a SVM model with default parameters

First, let's split the training and testing dataset. Training dataset will be used to train and tune models, and testing dataset will be used to evaluate the models. Note that you may also split a validation dataset from the training dataset for model tuning only.

[11]:
```
# First, let's split the training and testing dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪stratify=y, random_state = rs)
```

Okay, now we have the training and testing datasets ready, let's start the model training task.

We first define a `sklearn.svm import SVC` model with all default arguments.

[12]:
```
model = SVC()
```

Then train the model with training dataset:

[13]:
```
model.fit(X_train, y_train.values.ravel())
```

[13]:
```
SVC()
```

and make predictions

[14]:
```
preds = model.predict(X_test)
```

Here we defined a utility method to evaluate the model performance.

[15]:
```
def evaluate_metrics(yt, yp):
    results_pos = {}
    results_pos['accuracy'] = accuracy_score(yt, yp)
    precision, recall, f_beta, _ = precision_recall_fscore_support(yt, yp,␣
 ↪average='binary')
    results_pos['recall'] = recall
    results_pos['precision'] = precision
    results_pos['f1score'] = f_beta
    return results_pos
```

[16]:
```
evaluate_metrics(y_test, preds)
```

[16]:
```
{'accuracy': 0.9568062827225131,
 'recall': 0.7727272727272727,
 'precision': 0.8793103448275862,
 'f1score': 0.8225806451612904}
```

As we can see from the evaluation results above, the default SVM model achieves relatively good performance on this binary classification task. The overall accuracy is around `0.95` and the f1score is around `0.82`.

Now, you have easily built a SVM model with relatively good performance. Can we achieve better classification performance by customizing the model?

1.6 Train SVM with different regularization parameters and kernels

The `SVC` model provided by sklearn has two important arguments to be tuned: regularization parameter `C` and `kernel`.

The `C` argument is a regularization parameter.

- For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly.
- Conversely, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane misclassifies more points.

The `kernel` argument specifies the kernel to be used for transforming features to higher-dimensional spaces, some commonly used non-linear kernels are:

- `rbf`: Gaussian Radial Basis Function (RBF)
- `poly`: Polynomial Kernel
- `sigmoid`: Sigmoid Kernel

Let's first try `C = 10` and `kernel = 'rbf'`

[17]:
```
model = SVC(C=10, kernel='rbf')
model.fit(X_train, y_train.values.ravel())
```

```
preds = model.predict(X_test)
evaluate_metrics(y_test, preds)
```

[17]:
```
{'accuracy': 0.9679319371727748,
 'recall': 0.8333333333333334,
 'precision': 0.9116022099447514,
 'f1score': 0.870712401055409}
```

You should see that we have better performance than the default SVM model trained in the previous step. Now, it's your turn to try different parameters yourself.

1.6.1 Coding Exercise: Try different `C` values and `kernels` to see which combination produces SVM models with better classification performance.

[18]:
```
# Type your code here
model1 = SVC(C=100, kernel = "sigmoid")
model1.fit(X_train, y_train.values.ravel())
preds1 = model1.predict(X_test)
evaluate_metrics(y_test, preds1)
```

[18]:
```
{'accuracy': 0.8331151832460733,
 'recall': 0.3888888888888889,
 'precision': 0.36492890995260663,
 'f1score': 0.3765281173594132}
```

1.7 Tune regularization parameter C and Kernels via GridSearch

Exhaustively trying different hyperparameters by hands is infeasible. Thus, `sklearn` provides users with many automatic hyperparameter tuning methods. A popular one is grid search cross-validation `GridSearchCV`

Next, let's quickly try `GridSearchCV` to find the optimized `C` and `kernel` combination:

We first define some candidate parameter values we want to search in a `dict` object, like the following setting:

[19]:
```
params_grid = {
    'C': [1, 10, 100],
    'kernel': ['poly', 'rbf', 'sigmoid']
}
```

Then, we define a SVM model

[20]:
```
model = SVC()
```

and use create a `GridSearchCV` method to grid search `params_grid` and find the optimized combination with best `f1` score. The searching process may take several minutes to complete.

[21]:
```
# Define a GridSearchCV to search the best parameters
grid_search = GridSearchCV(estimator = model,
                           param_grid = params_grid,
                           scoring='f1',
                           cv = 5, verbose = 1)
# Search the best parameters with training data
grid_search.fit(X_train, y_train.values.ravel())
best_params = grid_search.best_params_
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
```

[22]:
```
best_params
```

[22]:
```
{'C': 100, 'kernel': 'rbf'}
```

Okay, we can see `C=100` and `kernel=`rbf" seems to produce the highest f1score. Let's quickly try this combination to see the model performance.

[23]:
```
model = SVC(C=100, kernel='rbf')
model.fit(X_train, y_train.values.ravel())
preds = model.predict(X_test)
evaluate_metrics(y_test, preds)
```

[23]:
```
{'accuracy': 0.9698952879581152,
 'recall': 0.8787878787878788,
 'precision': 0.8877551020408163,
 'f1score': 0.8832487309644671}
```

The best f1score now becomes `0.88` after hyperparameter tuning.

1.8 Plot SVM hyperplane and margin

Okay, you have learned how to define, train, evaluate, and fine-tune a SVM model with `sklearn`. However, so far we only obtained plain evaluation metrics and they are not intuitive to help us understand and interpret an SVM model. It would be great to visualize the see actual hyperplains and margins learned in an SVM model.

Since it is challenging to visualize a hyperplane higher than 3 dimensions. To illustrate the idea, we will focus on a 2-dimensional feature space.

We first simplify the dataset with only two features `Calories` and `Dietary Fiber`, and include only 1000 instances:

[24]:
```
simplified_food_df = food_df[['Calories', 'Dietary Fiber', 'class']]
```

[25]:
```
X = simplified_food_df.iloc[:1000, :-1].values
y = simplified_food_df.iloc[:1000, -1:].values
```

and we undersample the majority class `Class = 0` to balance the class distribution so we will produce a clearer visualization.

[26]:
```
under_sampler = RandomUnderSampler(random_state=123)
X_under, y_under = under_sampler.fit_resample(X, y)
```

[27]:
```
print(f"Dataset resampled shape, X: {X_under.shape}, y: {y_under.shape}")
```

```
Dataset resampled shape, X: (62, 2), y: (62,)
```

To better show the hyperplane and margins, we normalize the features with a `MinMaxScaler`.

[28]:
```
scaler = MinMaxScaler()
X_under = scaler.fit_transform(X_under)
```

Okay, let's first train a linear SVM model with `kernel=linear` so that we can get a linear hyperplane and margins.

[29]:
```
linear_svm = SVC(C=1000, kernel='linear')
linear_svm.fit(X_under, y_under)
```

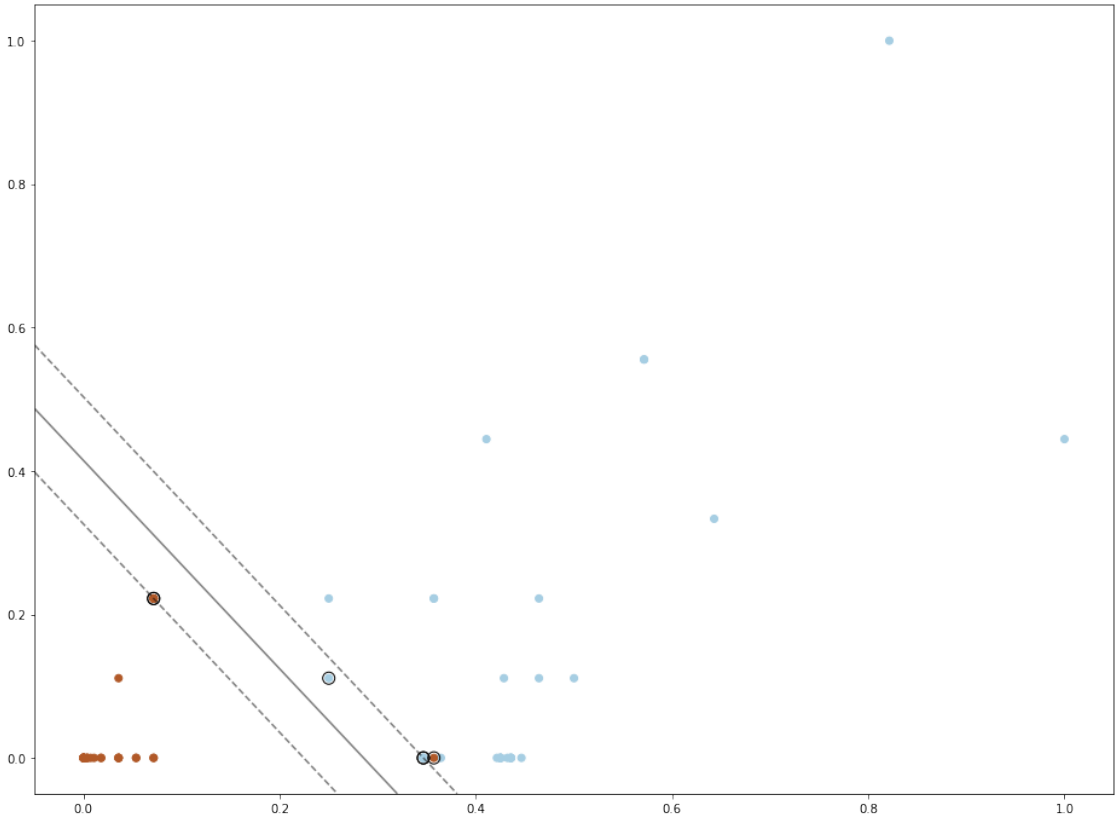
[29]:
```
SVC(C=1000, kernel='linear')
```

Here we also provided an utility method to plot the decision boundary (hyperplane), support vectors, and margins. You may write your own visualization method if you are interested.

[30]:
```
def plot_decision_boundry(X, y, model):
    plt.figure(figsize=(16, 12))
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)

    # plot the decision function
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # create grid to evaluate model
    xx = np.linspace(xlim[0], xlim[1], 30)
    yy = np.linspace(ylim[0], ylim[1], 30)
    YY, XX = np.meshgrid(yy, xx)
    xy = np.vstack([XX.ravel(), YY.ravel()]).T
    Z = model.decision_function(xy).reshape(XX.shape)

    # plot decision boundary and margins
    ax.contour(
        XX, YY, Z, colors="k", levels=[-1, 0, 1], alpha=0.5, linestyles=["--",␣
↪"-", "--"]
    )

    # plot support vectors
    ax.scatter(
        model.support_vectors_[:, 0],
```

```
        model.support_vectors_[:, 1],
        s=100,
        linewidth=1,
        facecolors="none",
        edgecolors="k",
    )
    plt.show()
```

[31]:
```
plot_decision_boundry(X_under, y_under, linear_svm)
```



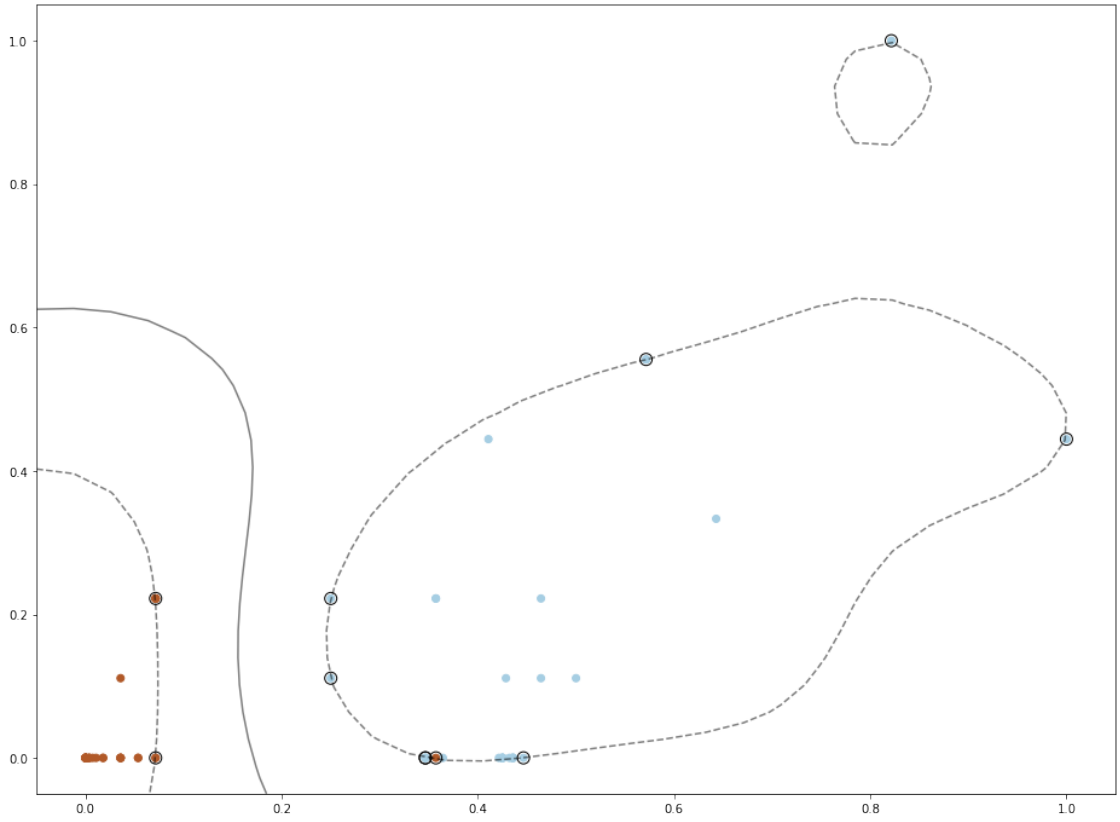
Okay, we can see a clear linear hyperplane seperates the two classes (Blue dots vs Orange dots). The highlighted dots are the support vectors determining the hyperplain.

If we want to include non-linear kernels, we should get a non-linear decision boundary in the 2-d space (maybe linear in higher feature space). So here we use a `rbf` kernel:

[32]:
```
svm_rbf_kernel = SVC(C=100, kernel='rbf')
svm_rbf_kernel.fit(X_under, y_under)
```

[32]:
```
SVC(C=100)
```

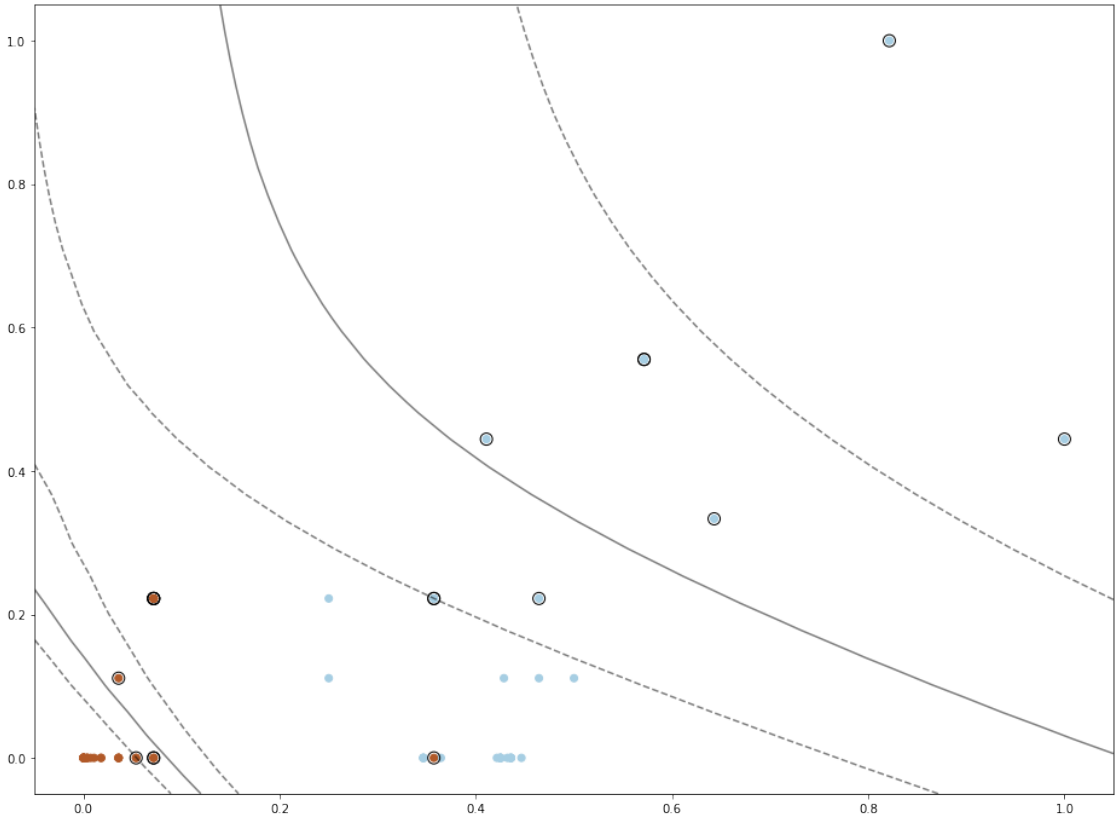
[33]:
```
plot_decision_boundry(X_under, y_under, svm_rbf_kernel)
```



We now see a non-linear hyperplane and margins separating the two classes.

1.8.1 Coding Exercise: Try different `C` values and `kernels` to see the how they affect the hyperplanes and margins.

[34]:
```
## Type your code here
svm_sig = SVC(C =1, kernel = 'sigmoid')
svm_sig.fit(X_under, y_under)
plot_decision_boundry(X_under, y_under, svm_sig)
```



Click here for a sample solution

```
svm_rbf_kernel = SVC(C=100, kernel='poly')
svm_rbf_kernel.fit(X_under, y_under)
plot_decision_boundry(X_under, y_under, svm_rbf_kernel)
```

1.9 Next Steps

Great! Now you have learned and practiced SVM model and applied it to solve a real-world food classification problem for diabetic patients. You also learned how to visualize the hyperplanes and margins generated by the SVM models.

Next, you will be learning other popular classification models with different structures, assumptions, cost functions, and application scenarios.

1.10 Authors

Yan Luo

1.10.1 Other Contributors

1.11 Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2021-9-23	1.0	Yan	Created the initial version
2022-2-9	1.1	Steve Hord	QA pass