ΕΛΛΗΝΙΚΟ
ΑΝΟΙΚΤΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ

Panagiotis Paltsokas
std163861@ac.eap.gr
#ΗΛΕ-48

# DAMA 61 : Written Assignment 2

## Problem 1

We are going to use different variants of regressors to model a sinusoidal function. Use the following code to create a set of non-linear data:

```
import numpy as np
np.random.seed(42)
m = 1000
X = 5 * np.random.rand(m, 1) - 2.5
y = np.sin(X)*100 + np.random.randn(m, 1)
```

1) Apply standardization to the data and plot the learning curve for Linear Regression.
2) Transform the data into a polynomial of degree 50, apply standardization and plot the learning curve.
3) Repeat the process for a Regularized Linear Regression model using Ridge Regression with alpha=0.001 and comment on the differences between the three plots.
4) Apply 10-fold cross-validation for the simple Linear Regression model and calculate the mean RMSE and its standard deviation.
5) Apply 10-fold cross-validation for the polynomial model without regularization and calculate the mean RMSE.
6) Apply 10-fold cross-validation for the regularized model and calculate the mean RMSE.
7) Comment on your results in the queries 4, 5, and 6.

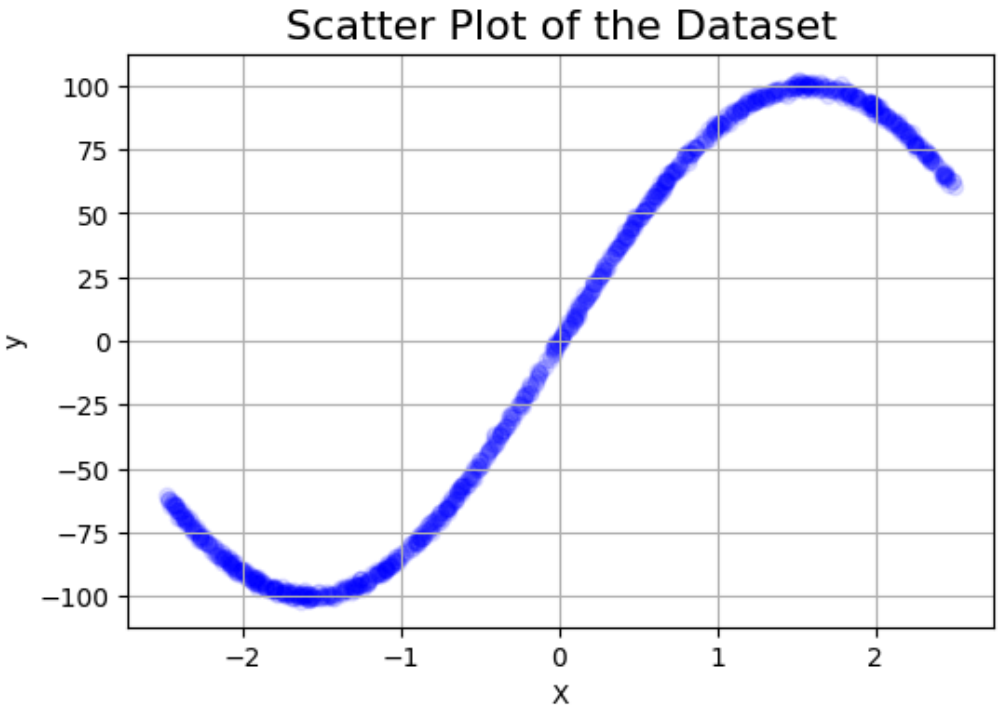
We are going to use different variants of regressors to model a sinusoidal function. Let's create a set of non-linear data:

In [1]:
```
import numpy as np
np.random.seed(42)
m = 1000
X = 5 * np.random.rand(m, 1) - 2.5
y = np.sin(X)*100 + np.random.randn(m, 1)
```

In [2]:
```
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import learning_curve, cross_val_score
```

Plotting a scatter plot to visualize the distribution of X and y values in the dataset.

In [3]:
```
plt.figure(figsize=(6, 4))
plt.scatter(X, y, color='b', alpha=0.1)
plt.title("Scatter Plot of the Dataset", fontsize=16)
plt.xlabel("X")
plt.ylabel("y")
plt.grid(True)
plt.show()
```



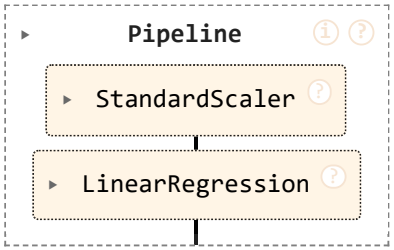
# 1.1

## Apply standardization to the data and plot the learning curve for Linear Regression.

Create a pipeline that first standardizes the data using StandardScaler, and then fits a Linear Regression model.

In [4]:
```
pipeline_linear = make_pipeline(StandardScaler(), LinearRegression())
pipeline_linear
```

Out[4]:




Compute learning curves using cross-validation (5-fold) and the negative RMSE as the scoring metric.

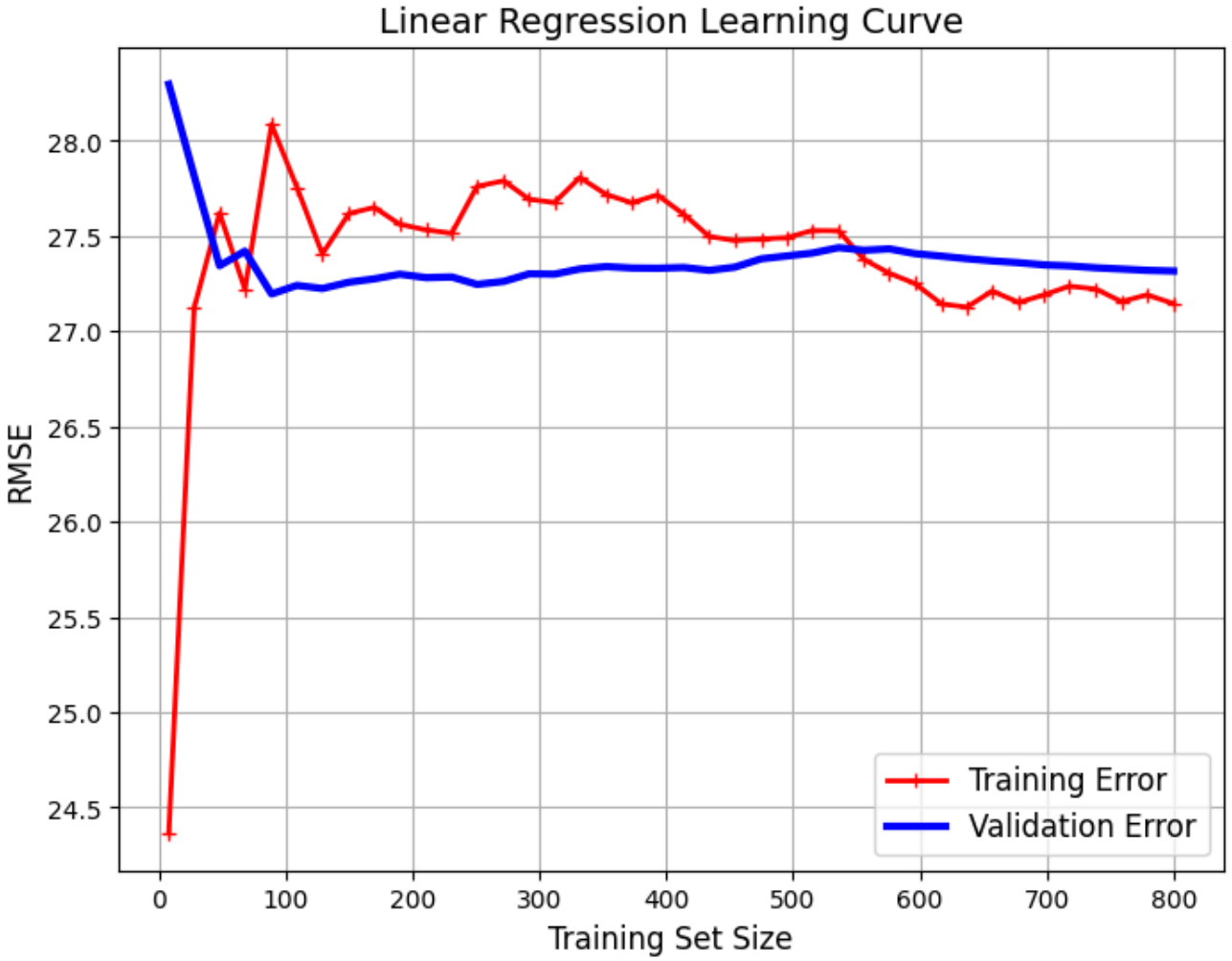
In [5]:
```
train_sizes, train_scores, valid_scores = learning_curve(pipeline_linear, X, y, train_sizes=np.linspace(0.01, 1.0, 40),
                                                         cv=5, scoring="neg_root_mean_squared_error")
# Negate the scores to get the RMSE for training and validation
train_errors= -train_scores.mean(axis=1)
valid_errors= -valid_scores.mean(axis=1)
```

In [6]:
```
print(f"Average training error (RMSE): {train_errors.mean():.3f} \nAverage validation error (RMSE) : {valid_errors.mean():.3f}")
```

```
Average training error (RMSE): 27.390
Average validation error (RMSE) : 27.369
```

Plot the learning curves for Linear Regression

In [7]:
```
plt.figure(figsize=(8,6))
plt.plot(train_sizes, train_errors, "r-+", linewidth=2, label="Training Error")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3, label="Validation Error")
plt.title("Linear Regression Learning Curve", fontsize=14)
plt.xlabel("Training Set Size",fontsize=12)
plt.ylabel("RMSE",fontsize=12)
plt.legend(loc='lower right',fontsize=12)
plt.grid(True)
plt.show()
```



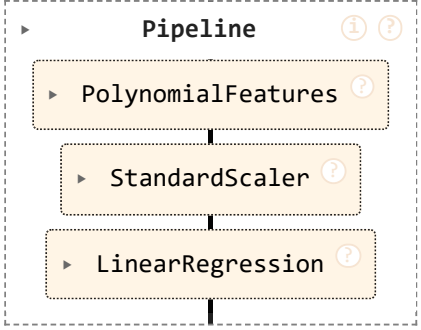

## 1.2

### Transform the data into a polynomial of degree 50, apply standardization and plot the learning curve.

Create a pipeline for Polynomial Regression that transforms data into polynomial features of degree 50, standardizes the features, and fits a linear model to the polynomial-transformed features.

In [8]:
```
pipeline_polynomial = make_pipeline(PolynomialFeatures(degree=50, include_bias=False), StandardScaler(), LinearRegression())
pipeline_polynomial
```

Out[8]:




Compute learning curves for the polynomial regression model, using cross-validation (5-fold) and the negative RMSE as the scoring metric.

In [9]:
```
train_sizes, train_scores, valid_scores = learning_curve(
    pipeline_polynomial, X, y, train_sizes=np.linspace(0.01, 1.0, 40), # training sizes range from 1% to 100% of the data in 40 steps.
    cv=5, scoring="neg_root_mean_squared_error")

# Again, negate the scores to get the RMSE for training and validation.
train_errors = -train_scores.mean(axis=1)
valid_errors = -valid_scores.mean(axis=1)
```

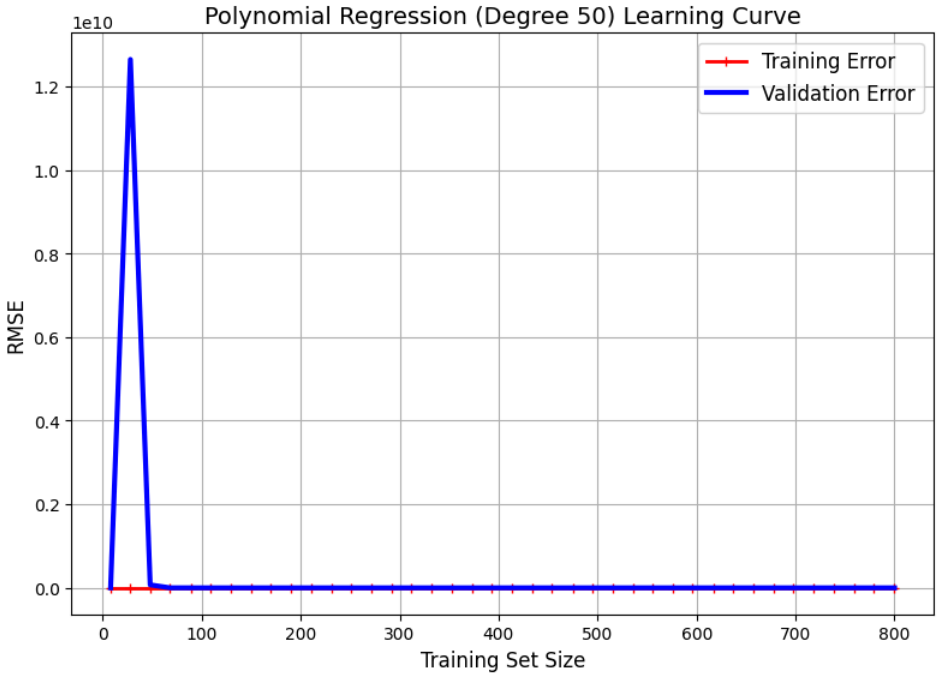
In [10]:
```
print(f"Average training error (RMSE): {train_errors.mean():.3f} \nAverage validation error: {valid_errors.mean():.3f}")
```

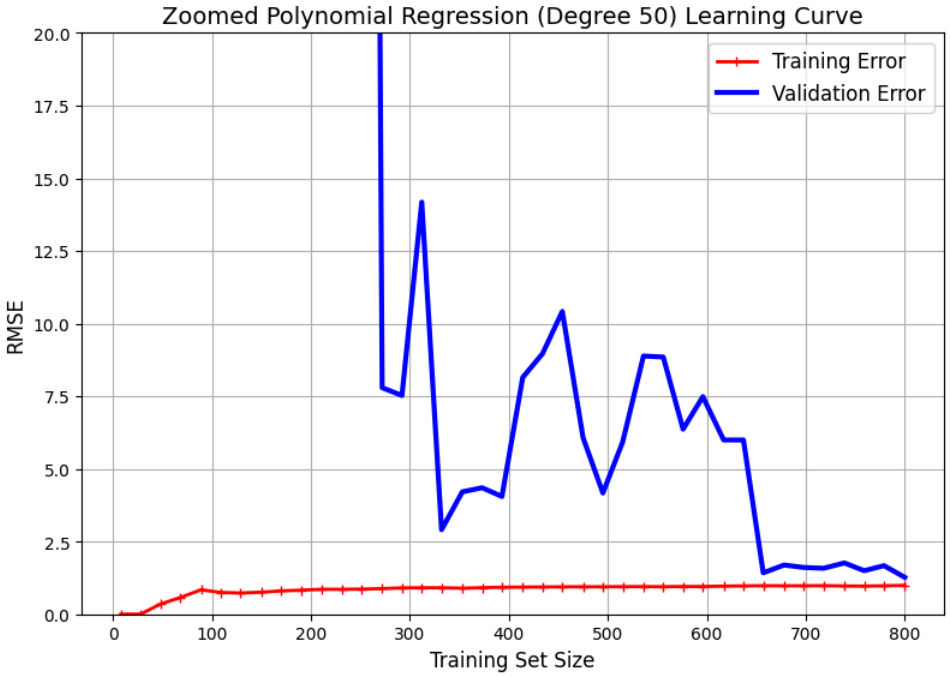
```
Average training error (RMSE): 0.845
Average validation error: 317887972.780
```

Plot learning curves for polynomial regression (degree 50)

In [11]:
```
plt.figure(figsize=(16, 6)) # Set figure size for dual subplots

plt.subplot(1, 2, 1)
plt.plot(train_sizes, train_errors, "r-+", linewidth=2, label="Training Error")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3, label="Validation Error")
plt.title("Polynomial Regression (Degree 50) Learning Curve", fontsize=14)
plt.xlabel("Training Set Size", fontsize=12)
plt.ylabel("RMSE", fontsize=12)
plt.legend(loc='upper right', fontsize=12)
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(train_sizes, train_errors, "r-+", linewidth=2, label="Training Error")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3, label="Validation Error")
plt.title("Zoomed Polynomial Regression (Degree 50) Learning Curve", fontsize=14)
plt.xlabel("Training Set Size", fontsize=12)
plt.ylabel("RMSE", fontsize=12)
plt.legend(loc='upper right', fontsize=12)
plt.grid(True)
plt.ylim(0, 20)
```

```
plt.tight_layout()
plt.show()
```



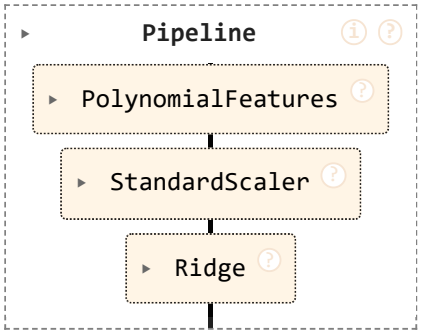

## 1.3

## Repeat the process for a Regularized Linear Regression model using Ridge Regression with alpha=0.001 and comment on the differences between the three plots.

Create a pipeline for Ridge regression with polynomial features of degree 50 and scaling.

In [12]:
```
pipeline_ridge = make_pipeline(PolynomialFeatures(degree=50, include_bias=False), StandardScaler(), Ridge(alpha=0.001))
pipeline_ridge
```

Out[12]:




Compute learning curves for the ridge regression pipeline.

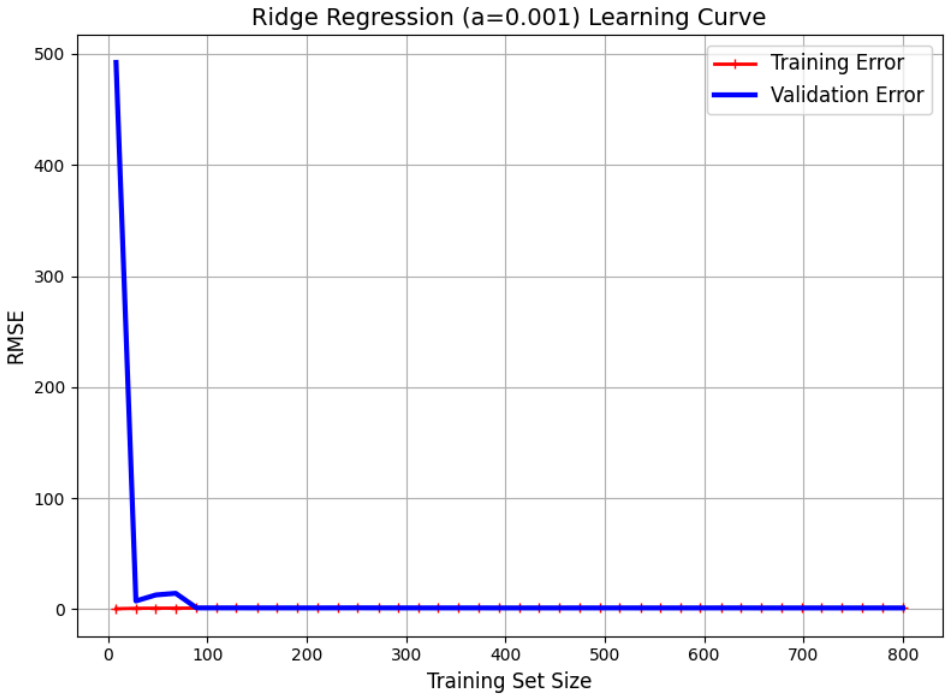
In [13]:
```
train_sizes, train_scores, valid_scores = learning_curve(pipeline_ridge, X, y, train_sizes=np.linspace(0.01, 1.0, 40),
                                                         cv=5, scoring="neg_root_mean_squared_error")
train_errors= -train_scores.mean(axis=1)
valid_errors= -valid_scores.mean(axis=1)
```

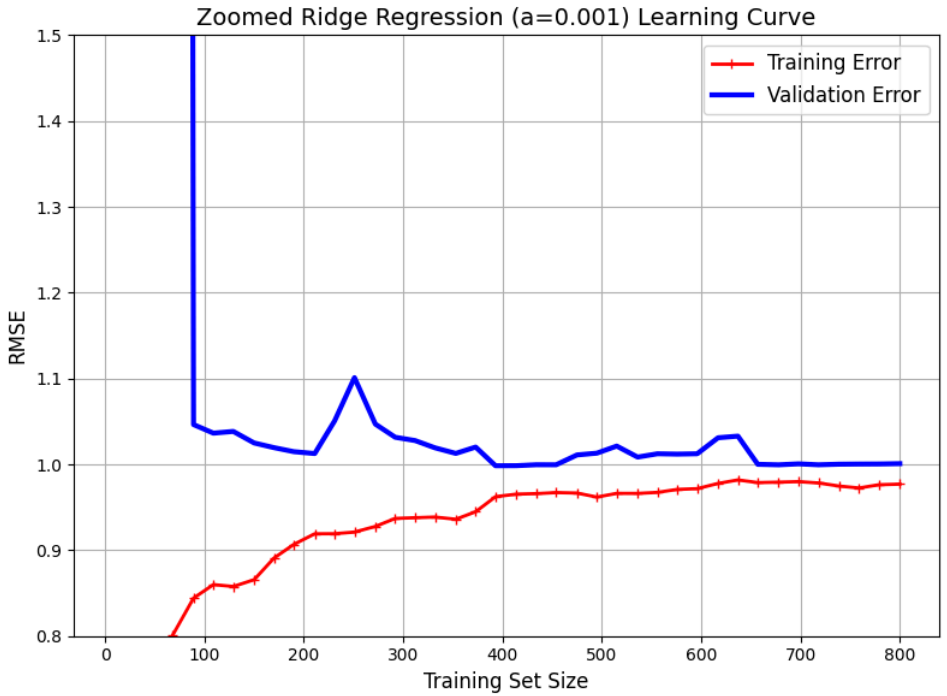
In [14]:
```
print("Average training error (RMSE):", round(train_errors.mean(),3), "\nAverage validation error:", round(valid_errors.mean(),3))
```

```
Average training error (RMSE): 0.913
Average validation error: 14.08
```

Plot the learning curves for the ridge regression model.

In [15]:
```
plt.figure(figsize=(16, 6))

plt.subplot(1, 2, 1)
plt.plot(train_sizes, train_errors, "r-+", linewidth=2, label="Training Error")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3, label="Validation Error")
plt.title("Ridge Regression (a=0.001) Learning Curve", fontsize=14)
plt.xlabel("Training Set Size", fontsize=12)
plt.ylabel("RMSE", fontsize=12)
plt.legend(loc='upper right', fontsize=12)
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(train_sizes, train_errors, "r-+", linewidth=2, label="Training Error")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3, label="Validation Error")
plt.title("Zoomed Ridge Regression (a=0.001) Learning Curve", fontsize=14)
plt.xlabel("Training Set Size", fontsize=12)
plt.ylabel("RMSE", fontsize=12)
plt.legend(loc='upper right', fontsize=12)
plt.ylim(0.8,1.5)
plt.grid(True)

plt.tight_layout()
plt.show()
```




### Comment on Linear Regression (1.1) :

Based on the learning curve, we can observe that the training error starts very low with a small training set, as the model is able to memorize the data points. As the training set size increases, the training error rises and stabilizes. The validation error starts high but quickly stabilizes as more data is used for training. However, the validation error remains fairly constant, showing that the model does not generalize better beyond a certain point. Both errors are relatively high, which indicates high bias, and end up at a plateau, fairly close to each other. This is typical of a model that is underfitting and is not complex enough to accurately capture the underlying relationship in the data

### Comment on Polynomial Regression (1.2) :

The learning curve reveals that the training error is nearly zero, indicating that the model is highly complex and fits the training data perfectly. However, the validation error is substantially higher, highlighting a clear gap between the training and validation performance. This behavior is characteristic of a heavily overfit model, which is commonly observed with high-degree polynomials. The model is too flexible and ends up fitting the noise in the training set, leading to poor generalization performance.

### Comment on Ridge Regression (1.3) :

The ridge regression with polynomial features helps reduce overfitting by adding regularization. The training error is higher compared to polynomial regression without regularization, but the validation error is significantly lower, indicating better generalization due to the regularization provided by ridge.The regularization term penalizes large coefficients, which prevents the model from fitting the noise and reduces variance, ultimately leading to a model that generalizes better to unseen data.

## 1.4

## Apply 10-fold cross-validation for the simple Linear Regression model and calculate the mean RMSE and its standard deviation.

Perform 10-fold cross-validation for the linear pipeline.

In [16]:
```
scores = cross_val_score(pipeline_linear, X, y, scoring = "neg_root_mean_squared_error", cv=10)
average_rmse = -scores.mean()
std_rmse = scores.std()
```

In [17]:
```
print(f"Mean RMSE (Linear Regression): {average_rmse:.3f}\nStandard Deviation: {std_rmse:.3f}")
```

```
Mean RMSE (Linear Regression): 27.241
Standard Deviation: 1.089
```

## 1.5

## Apply 10-fold cross-validation for the polynomial model without regularization and calculate the mean RMSE.

Perform 10-fold CV for the polynomial model without regularization.

In [18]:
```
scores = cross_val_score(pipeline_polynomial, X, y, scoring="neg_root_mean_squared_error", cv=10)
average_rmse = -scores.mean()
```

In [19]:
```
print(f"Mean RMSE (Polynomial Regression without Regularization): {average_rmse:.3f}")
```

```
Mean RMSE (Polynomial Regression without Regularization): 1.522
```

## 1.6

## Apply 10-fold cross-validation for the regularized model and calculate the mean RMSE.

Perform 10-fold CV for the ridge regularized model.

In [20]:
```
scores = cross_val_score(pipeline_ridge, X, y, scoring="neg_root_mean_squared_error", cv=10)
average_rmse = -scores.mean()
```

In [21]:
```
print(f"Mean RMSE (Regularized Ridge Regression): {average_rmse:.3f}")
```

```
Mean RMSE (Regularized Ridge Regression): 0.994
```

**The linear regression** has a high RMSE which indicates that the model underfits the data, and with a low standard deviation, it consistenly remains in the high error, reflectin the model's inability to generalize well to the data's complexity.
**Introducing the polynomial features** of degree 50, greatly reduces the RMSE, and the model can now capture the non linear relationships in the data. However the lack of regularization might make the model sensitive to noise in the data, and the high degree of the polynomial bears the risk of not generalizing well to new data.
**Applying Ridge regularization** further reduces the RMSE and now achieves the best performance among the three models. Ridge regression penalizes large coefficients (excessive complexity), helping prevent overfitting.

# Problem 2

## Problem 2

Apply linear and RBF SVMs on the Breast Cancer dataset.

1) Download the Breast Cancer dataset from the `sklearn.datasets` package.
2) Out of the 30 available features, select only "Worst area" and "Mean Concave Points".
3) Visualize the data from the Positive and Negative class with the "Mean Concave Points" on the x-axis and "Worst Area" on the y-axis.
4) Train two linear SVM classifiers with the regularization hyperparameter C equal to 0.1 and 1000, respectively. *Don't forget data standardization.*
5) Plot the data points, the decision boundaries and the margins for the two classifiers.
6) Display the number of Support Vectors and the F1-score for each of the two classifiers.
7) Run a Grid Search for an RBF SVM, with the following hyperparameter options:
**C:** [0.1, 1, 10, 100] , **gamma:** [0.1, 1, 10, 100]
8) Display the best hyperparameter values, the number of support vectors, and the F1-score for the best model.
9) Plot the data points and the decision boundary for the best RBF model.

## 2.1

### Load the Breast Cancer dataset fro mthe sklearn.datasets package.

Load the necessary libraries for the project. (Cell was updated as the solution was progressing)

In [22]:
```
from sklearn import datasets
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from sklearn.pipeline import make_pipeline
from sklearn.svm import LinearSVC, SVC
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import f1_score
from sklearn.model_selection import GridSearchCV
```

Load the dataset

In [23]:
```
cancer = datasets.load_breast_cancer() # Load the breast cancer dataset directly from sklearn
X = cancer.data # Extract the features
y = cancer.target # Extract the target variable
```

# 2.2

## Out of the 30 available features, select only "Worst area" and "Mean Concave Points".

Convert the dataset into a dataframe to perform EDA for better data understanding

In [24]:
```
cancer_df = pd.DataFrame(cancer.data, columns=cancer.feature_names)
cancer_df['target']=y # Create a new column for the target
cancer_df
```

Out[24]:

| | mean radius | mean texture | mean perimeter | mean area | mean smoothness | mean compactness | mean concavity | mean concave points | mean symmetry | mean fractal dimension | ... | worst texture | worst perimeter | worst area | worst smoothness |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 17.99 | 10.38 | 122.80 | 1001.0 | 0.11840 | 0.27760 | 0.30010 | 0.14710 | 0.2419 | 0.07871 | ... | 17.33 | 184.60 | 2019.0 | 0.16220 |
| **1** | 20.57 | 17.77 | 132.90 | 1326.0 | 0.08474 | 0.07864 | 0.08690 | 0.07017 | 0.1812 | 0.05667 | ... | 23.41 | 158.80 | 1956.0 | 0.12380 |
| **2** | 19.69 | 21.25 | 130.00 | 1203.0 | 0.10960 | 0.15990 | 0.19740 | 0.12790 | 0.2069 | 0.05999 | ... | 25.53 | 152.50 | 1709.0 | 0.14440 |
| **3** | 11.42 | 20.38 | 77.58 | 386.1 | 0.14250 | 0.28390 | 0.24140 | 0.10520 | 0.2597 | 0.09744 | ... | 26.50 | 98.87 | 567.7 | 0.20980 |
| **4** | 20.29 | 14.34 | 135.10 | 1297.0 | 0.10030 | 0.13280 | 0.19800 | 0.10430 | 0.1809 | 0.05883 | ... | 16.67 | 152.20 | 1575.0 | 0.13740 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **564** | 21.56 | 22.39 | 142.00 | 1479.0 | 0.11100 | 0.11590 | 0.24390 | 0.13890 | 0.1726 | 0.05623 | ... | 26.40 | 166.10 | 2027.0 | 0.14100 |
| **565** | 20.13 | 28.25 | 131.20 | 1261.0 | 0.09780 | 0.10340 | 0.14400 | 0.09791 | 0.1752 | 0.05533 | ... | 38.25 | 155.00 | 1731.0 | 0.11660 |
| **566** | 16.60 | 28.08 | 108.30 | 858.1 | 0.08455 | 0.10230 | 0.09251 | 0.05302 | 0.1590 | 0.05648 | ... | 34.12 | 126.70 | 1124.0 | 0.11390 |
| **567** | 20.60 | 29.33 | 140.10 | 1265.0 | 0.11780 | 0.27700 | 0.35140 | 0.15200 | 0.2397 | 0.07016 | ... | 39.42 | 184.60 | 1821.0 | 0.16500 |
| **568** | 7.76 | 24.54 | 47.92 | 181.0 | 0.05263 | 0.04362 | 0.00000 | 0.00000 | 0.1587 | 0.05884 | ... | 30.37 | 59.16 | 268.6 | 0.08996 |

569 rows × 31 columns

Print all the features to spot the "worst area" and "mean concave points" features' index

In [25]:
```
print(cancer_df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 31 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   mean radius              569 non-null    float64
 1   mean texture             569 non-null    float64
 2   mean perimeter           569 non-null    float64
 3   mean area                569 non-null    float64
 4   mean smoothness          569 non-null    float64
 5   mean compactness         569 non-null    float64
 6   mean concavity           569 non-null    float64
 7   mean concave points      569 non-null    float64
 8   mean symmetry            569 non-null    float64
 9   mean fractal dimension   569 non-null    float64
 10  radius error             569 non-null    float64
 11  texture error            569 non-null    float64
 12  perimeter error          569 non-null    float64
 13  area error               569 non-null    float64
 14  smoothness error         569 non-null    float64
 15  compactness error        569 non-null    float64
 16  concavity error          569 non-null    float64
 17  concave points error     569 non-null    float64
 18  symmetry error           569 non-null    float64
 19  fractal dimension error  569 non-null    float64
 20  worst radius             569 non-null    float64
 21  worst texture            569 non-null    float64
 22  worst perimeter          569 non-null    float64
 23  worst area               569 non-null    float64
 24  worst smoothness         569 non-null    float64
 25  worst compactness        569 non-null    float64
 26  worst concavity          569 non-null    float64
 27  worst concave points     569 non-null    float64
 28  worst symmetry           569 non-null    float64
 29  worst fractal dimension  569 non-null    float64
 30  target                   569 non-null    int32  
dtypes: float64(30), int32(1)
memory usage: 135.7 KB
None
```

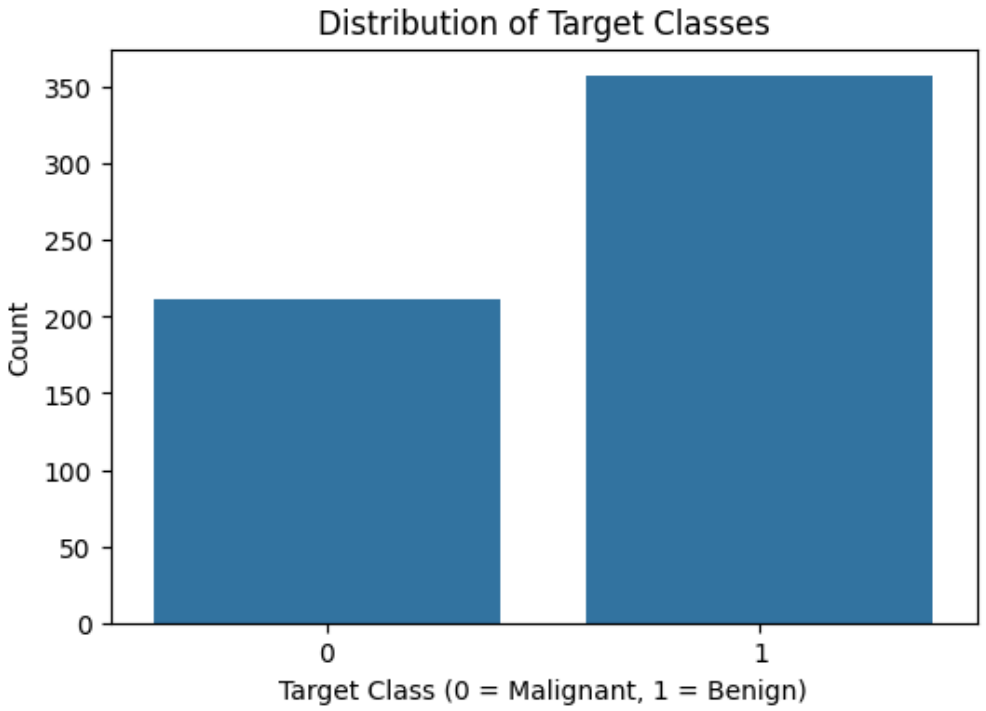
Check the distribution of the target variable

In [26]:
```
print(cancer_df['target'].value_counts())
```

```
target
1    357
0    212
Name: count, dtype: int64
```

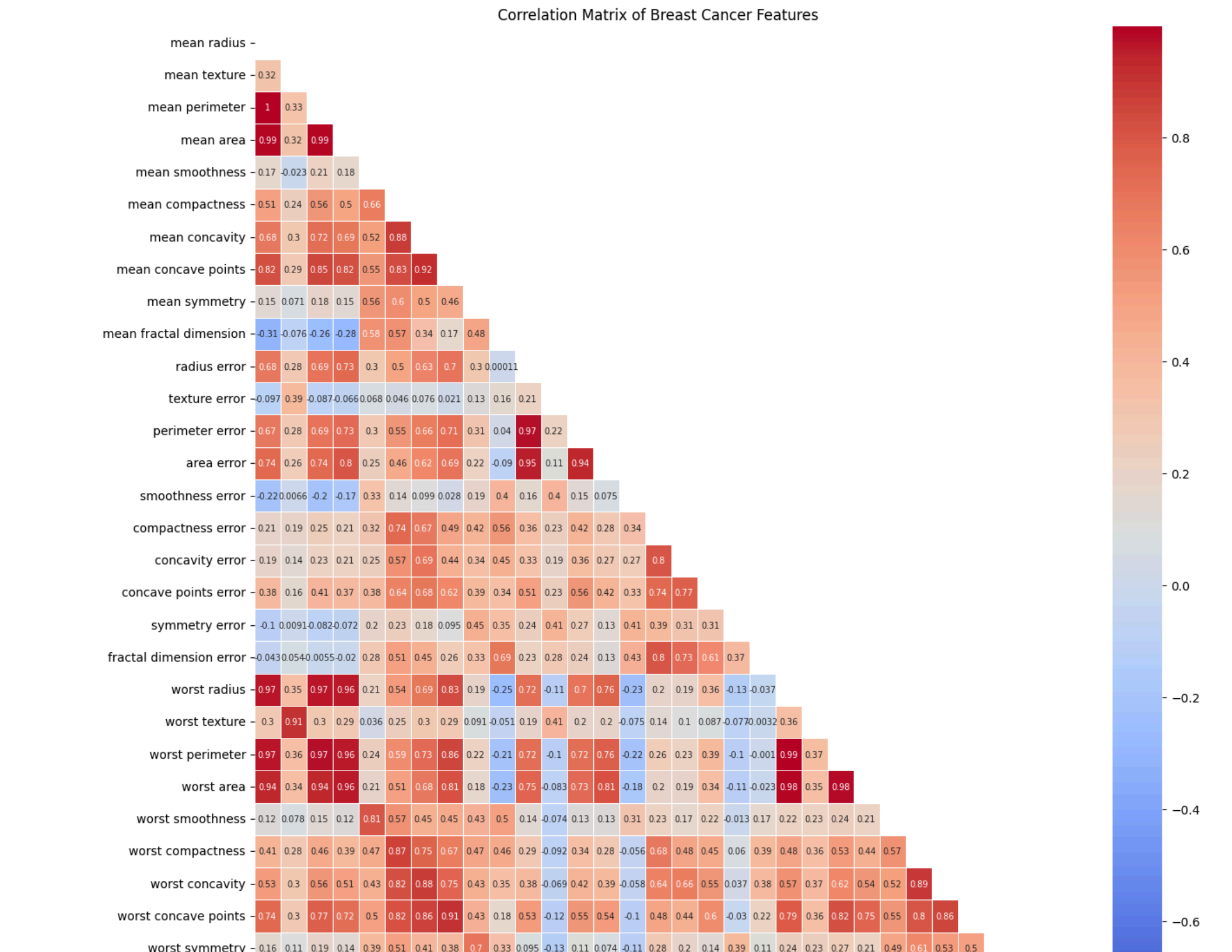
In [27]:
```
plt.figure(figsize=(6,4))
sns.countplot(x='target', data=cancer_df)
```

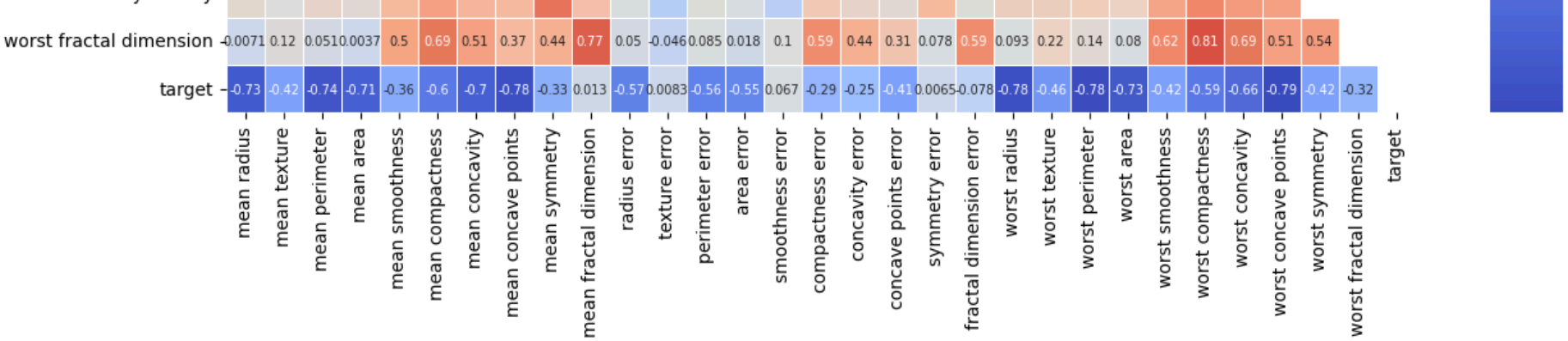
```
plt.title("Distribution of Target Classes")
plt.xlabel("Target Class (0 = Malignant, 1 = Benign)")
plt.ylabel("Count")
plt.show()
```




Create a correlation matrix

In [28]:
```
mask = np.triu(np.ones_like(cancer_df.corr(), dtype=bool)) # Create a mask to hide the upper triangle and make the heatmap more readable

plt.figure(figsize=(15, 15))
mask = np.triu(np.ones_like(cancer_df.corr(), dtype=bool))
sns.heatmap(cancer_df.corr(), mask=mask, annot=True, cmap='coolwarm', linewidths=0.5, annot_kws={"size": 7})
plt.title("Correlation Matrix of Breast Cancer Features")
plt.show()
```


Correlation Matrix of Breast Cancer Features
mean radius
mean texture
mean perimeter
mean area
mean smoothness
mean compactness
mean concavity
mean concave points
mean symmetry
mean fractal dimension
radius error
texture error
perimeter error
area error
smoothness error
compactness error
concavity error
concave points error
symmetry error
fractal dimension error
worst radius
worst texture
worst perimeter
worst area
worst smoothness
worst compactness
worst concavity
worst concave points
worst symmetry
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6




Select the features required for the problem.

In [29]:
```
X_selected = X[:, [7, 23]]
```

Again, create a dataframe for the two variables for better readability

In [30]:
```
X_selected_df = pd.DataFrame(X_selected, columns=['Worst Area', 'Mean Concave Points'])
X_selected_df['target'] = y
```

In [31]:
```
X_selected_df
```

Out[31]:

| | Worst Area | Mean Concave Points | target |
|---|---|---|---|
| **0** | 0.14710 | 2019.0 | 0 |
| **1** | 0.07017 | 1956.0 | 0 |
| **2** | 0.12790 | 1709.0 | 0 |
| **3** | 0.10520 | 567.7 | 0 |
| **4** | 0.10430 | 1575.0 | 0 |
| **...** | ... | ... | ... |
| **564** | 0.13890 | 2027.0 | 0 |
| **565** | 0.09791 | 1731.0 | 0 |
| **566** | 0.05302 | 1124.0 | 0 |
| **567** | 0.15200 | 1821.0 | 0 |
| **568** | 0.00000 | 268.6 | 1 |

569 rows × 3 columns

In [32]:
```
X_selected_df.loc[X_selected_df.iloc[:,2] == 0, X_selected_df.columns[[0]]]
```

Out[32]:

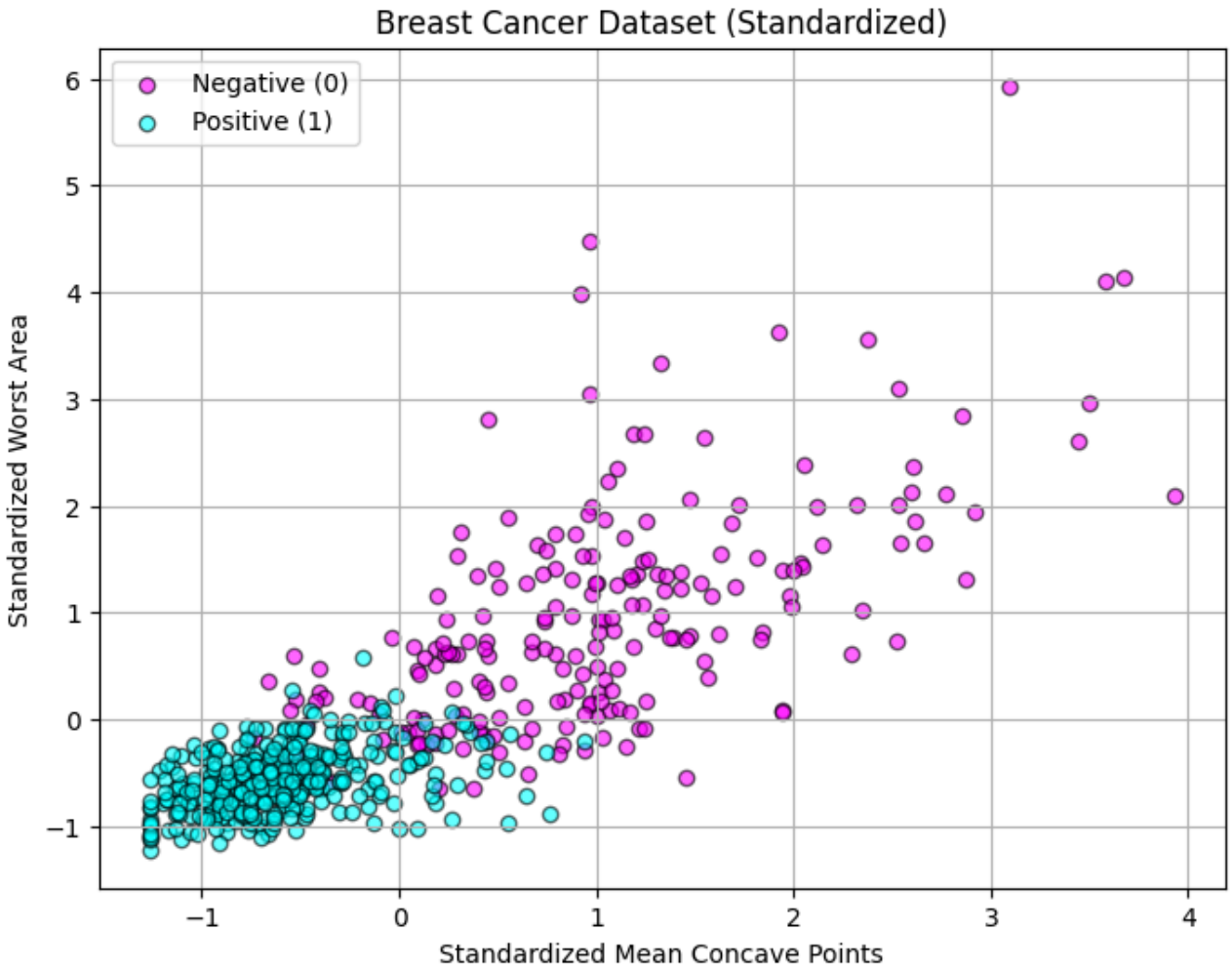
| | Worst Area |
|---|---|
| **0** | 0.14710 |
| **1** | 0.07017 |
| **2** | 0.12790 |
| **3** | 0.10520 |
| **4** | 0.10430 |
| **...** | ... |
| **563** | 0.14740 |
| **564** | 0.13890 |
| **565** | 0.09791 |
| **566** | 0.05302 |
| **567** | 0.15200 |

212 rows × 1 columns

## 2.3

## Visualize the data from the Positive and Negative class with the "Mean Concave Points" on the x-axis and "Worst Area" on the y-axis.

In [33]:
```
X_scaled = StandardScaler().fit_transform(X_selected) # Standardize the selected features so that mean = 0 and std = 1

# X_scaled[y == 0, 0] Worst Area malignant items (scaled)
# X_scaled[y == 0, 1] Mean Concave malignant benign items (scaled)
# X_scaled[y == 1, 0] Worst Area benign items (scaled)
# X_scaled[y == 1, 1] Mean Concave Points benign items (scaled)
plt.figure(figsize=(8, 6))
plt.scatter(X_scaled[y == 0, 0], X_scaled[y == 0, 1], c="magenta", label="Negative (0)", alpha=0.6, edgecolors="k")
plt.scatter(X_scaled[y == 1, 0], X_scaled[y == 1, 1], c="cyan", label="Positive (1)", alpha=0.6, edgecolors="k")
plt.xlabel("Standardized Mean Concave Points")
plt.ylabel("Standardized Worst Area")
plt.title("Breast Cancer Dataset (Standardized)")
plt.legend()
plt.grid(True)
plt.show()
```



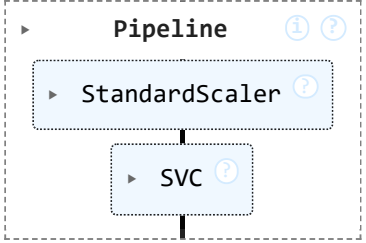

## 2.4

### Train two linear SVM classifiers with the regularization hyperparameter C equal to 0.1 and 1000, respectively.

Create and fit two separate SVM pipelines with a linear kernel, to first scale our features and then apply the SVC.
For the first classifier C=0.1, which means less penalty on margin violations. This means that the model may underfit.

In [34]:
```
clf_1 = make_pipeline(StandardScaler(), SVC(kernel="linear", C=0.1, random_state=42))
clf_1.fit(X_selected, y)
```

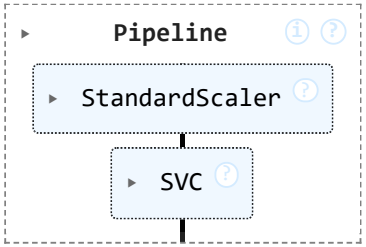
Out[34]:




For the second classifier C=1000, which means higher penalty on margin violations. That means that the model will try harder to correctly classify each training example, potentially leading to overfitting.

In [35]:
```
clf_2 = make_pipeline(StandardScaler(), SVC(kernel="linear", C=1000, random_state=42))
clf_2.fit(X_selected, y)
```

Out[35]:




## 2.5

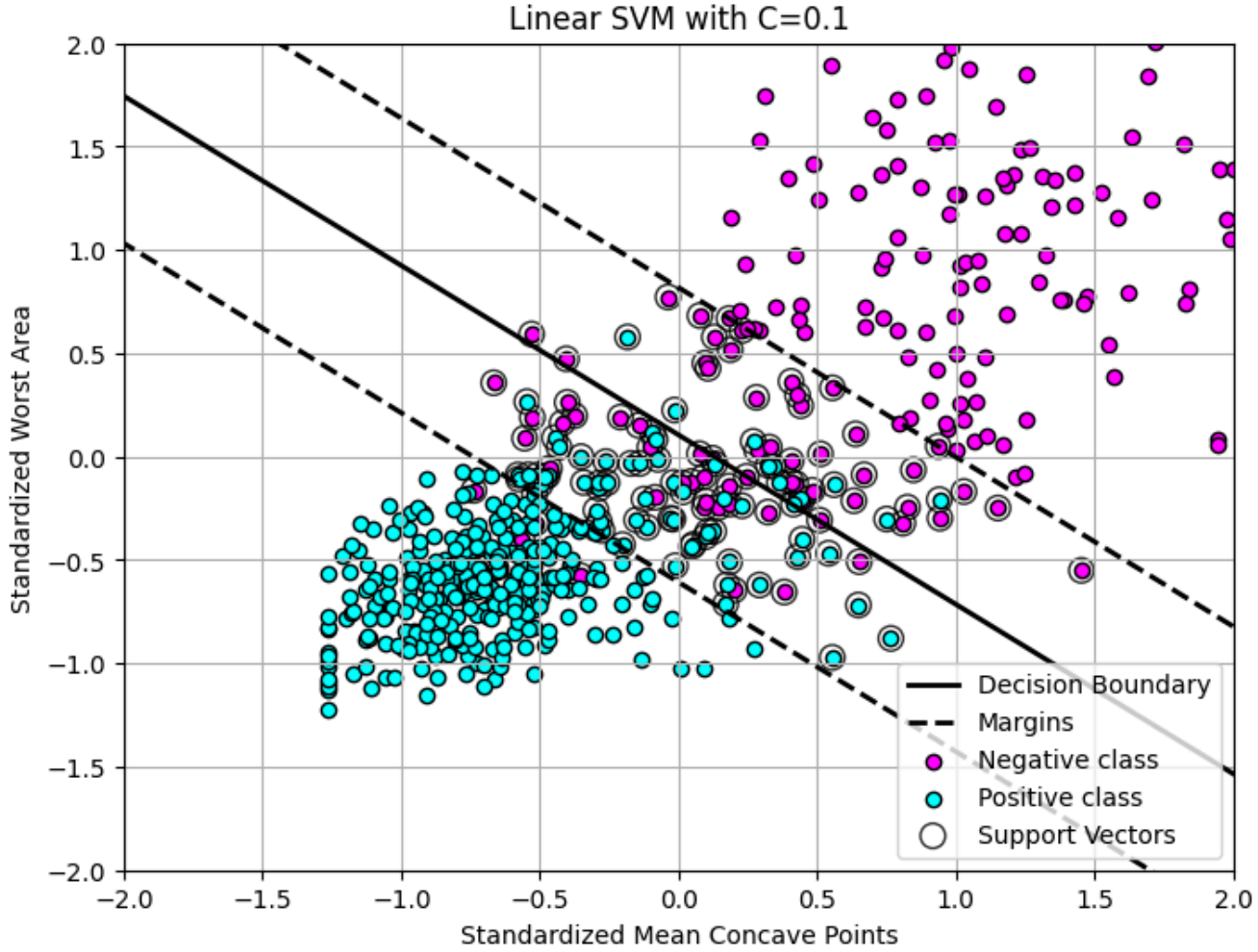
## Plot data points, decision boundaries, and margins for the two classifiers.

Define a function that will plot the decision boundary, margins, support vectors, and data points for a linear SVM classifier, and prints out the equations of the decision boundary and margin lines for reference.

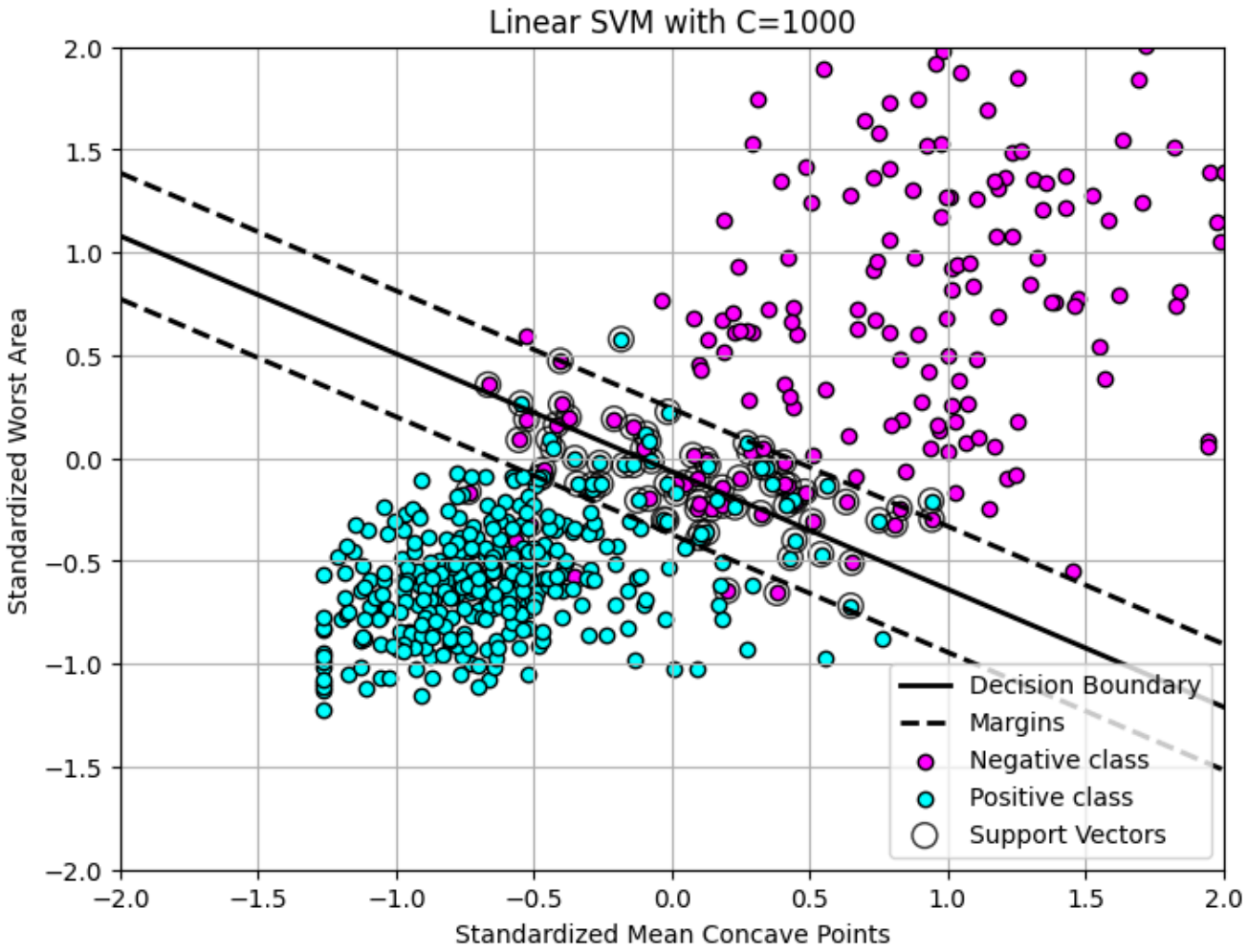
In [36]:
```
def plot_svc_decision_boundary(svm_clf, X, y, title):
    """
    Plots the decision boundary, margins, support vectors, and data points for a linear SVM classifier.
    Parameters:
    svm_clf : A trained SVM pipeline
    X : Array of shape (n_samples, 2) that holds the unscaled input data
    y : Array of shape (n_samples,1) that holds the class labels corresponding to each sample in X.
    title : str type, which will be the title of the plot
    Returns:
    None
    """
    # Extract the scaler and SVM model from the pipeline
    scaler = svm_clf.named_steps['standardscaler']
    svc = svm_clf.named_steps['svc']

    # Scale the input data X
    X_scaled = scaler.transform(X)
```

```
    # Compute the decision boundary
    w = svc.coef_[0]
    b = svc.intercept_[0]

    # Generate an array of 200 evenly spaced with a range from the min value to the max value of the "Mean Concave Points" data.
    # We start slightly below the minimum value, and we end slightly above the maximum value to ensure all data points
    # are within the grid
    x0 = np.linspace(X_scaled[:, 0].min() - 1, X_scaled[:, 0].max() + 1, 200)

    decision_boundary = -w[0] / w[1] * x0 - b / w[1]
    margin = 1 / np.linalg.norm(w)
    gutter_up = -w[0] / w[1] * x0 - (b - 1) / w[1]
    gutter_down = -w[0] / w[1] * x0 - (b + 1) / w[1]

    # Plot the decision boundary, and the margin lines
    plt.figure(figsize=(8, 6))
    plt.plot(x0, decision_boundary, 'k-', linewidth=2, label='Decision Boundary')
    plt.plot(x0, gutter_up, 'k--', linewidth=2, label='Margins')
    plt.plot(x0, gutter_down, 'k--', linewidth=2)

    # Plot the data points
    plt.scatter(X_scaled[y == 0, 0], X_scaled[y == 0, 1], color='magenta', label='Negative class', edgecolors='k')
    plt.scatter(X_scaled[y == 1, 0], X_scaled[y == 1, 1], color='cyan', label='Positive class', edgecolors='k')

    # Plot support vectors
    plt.scatter(svc.support_vectors_[:, 0], svc.support_vectors_[:, 1], s=100,
                facecolors='none', edgecolors='k', alpha=0.8, linewidths=1, label='Support Vectors', zorder=-2)

    plt.xlabel("Standardized Mean Concave Points")
    plt.ylabel("Standardized Worst Area")
    plt.xlim(-2,2)
    plt.ylim(-2,2)
    plt.title(title)
    plt.legend(loc='lower right')
    plt.grid(True)
    plt.show()

    # Print out the equations of the decision boundary, and the margin lines
    slope = -w[0] / w[1]
    intercept = -b / w[1]
    intercept_up = intercept + margin
    intercept_down = intercept - margin
    print(f"{title}")
    print(f"Decision Boundary: x1 = {slope:.4f} * x0 + {intercept:.4f}")
    print(f"Upper Margin: x1 = {slope:.4f} * x0 + {intercept_up:.4f}")
    print(f"Lower Margin: x1 = {slope:.4f} * x0 + {intercept_down:.4f}")
```

In [37]:
```
plot_svc_decision_boundary(clf_1, X_selected, y, title="Linear SVM with C=0.1")
plot_svc_decision_boundary(clf_2, X_selected, y, title="Linear SVM with C=1000")
```




```
Linear SVM with C=0.1
Decision Boundary: x1 = -0.8200 * x0 + 0.1052
Upper Margin: x1 = -0.8200 * x0 + 0.6563
Lower Margin: x1 = -0.8200 * x0 + -0.4459
```




```
Linear SVM with C=1000
Decision Boundary: x1 = -0.5727 * x0 + -0.0644
Upper Margin: x1 = -0.5727 * x0 + 0.2014
Lower Margin: x1 = -0.5727 * x0 + -0.3302
```

## 2.6

## Display the number of Support Vectors and the F1-score for each of the two classifiers.

Extract the SVC objects from the pipelines, to be able to access the support_vectors_ attribute.

In [38]:
```
svc_1 = clf_1.named_steps['svc']
svc_2 = clf_2.named_steps['svc']
svc_1.support_vectors_.shape, svc_2.support_vectors_.shape # Check the shape of the support vectors arrays.
```

Out[38]:
```
((129, 2), (85, 2))
```

Count the number of support vectors for each classifier.

In [39]:
```
num_support_vectors_1 = svc_1.support_vectors_.shape[0]
num_support_vectors_2 = svc_2.support_vectors_.shape[0]

print(f"There are {num_support_vectors_1} support vectors for classifier with C=0.1")
print(f"There are {num_support_vectors_2} support vectors for classifier with C=1000")
```

```
There are 129 support vectors for classifier with C=0.1
There are 85 support vectors for classifier with C=1000
```

Compute the F1-scores for each classifier.

In [40]:
```
# Predict on the training data
y_pred_1 = clf_1.predict(X_selected)
y_pred_2 = clf_2.predict(X_selected)

# Calculate the F1-score for each classifier
f1_score_1 = f1_score(y, y_pred_1)
f1_score_2 = f1_score(y, y_pred_2)
print(f"F1-score for classifier with C=0.1: {f1_score_1}")
print(f"F1-score for classifier with C=1000: {f1_score_2}")
```

```
F1-score for classifier with C=0.1: 0.9440654843110505
F1-score for classifier with C=1000: 0.9511854951185496
```

# 2.7

## Run a Grid Search for an RBF SVM, with the following hyperparameter options:

## C$:[0.1, 1, 10, 100]$, gamma: $[0.1, 1, 10, 100]$

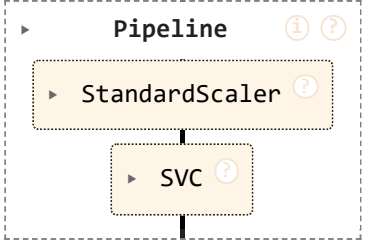
Define the hyperparameter options.

In [41]:
```
param_grid = {'svc__C': [0.1, 1, 10, 100], 'svc__gamma': [0.1, 1, 10, 100]}
```

Create a pipeline that scales the data and then applies a SVM with an RBF kernel.

In [42]:
```
rbf_pipeline = make_pipeline(StandardScaler(), SVC(kernel = 'rbf', random_state=42))
rbf_pipeline
```

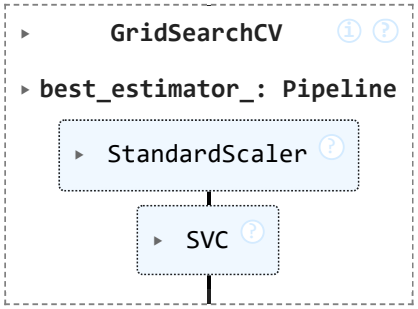
Out[42]:




Perform grid search with a 5-fold cross-validation setup, using the F1-score as the socring metric.

In [43]:
```
grid_search = GridSearchCV(rbf_pipeline, param_grid, cv=5, scoring='f1')
grid_search.fit(X_selected, y)
```

Out[43]:




## 2.8

## Display the best hyperparameter values, the number of support vectors, and the F1-score for the best model.

Extract the best model found by the grid search.

In [44]:
```
best_model = grid_search.best_estimator_ # best_estimator_ returns the pipeline with the best combination of parameters.
best_params = grid_search.best_params_ # best_params_ returns the parameters of the best pipeline.
best_svc = best_model.named_steps['svc'] # Extract the SVC model from the best pipeline.
num_support_vectors = best_svc.support_vectors_.shape[0] # Get the number of support vectors
best_f1_score = f1_score(y, best_model.predict(X_selected)) # Get the F1-score by evaluating the best model on the full dataset.

print(f"Best Model Parameters: {best_params}")
print(f"Number of Support Vectors for Best Model: {num_support_vectors}")
print(f"F1-Score for Best Model: {best_f1_score:.4f}")
```

```
Best Model Parameters: {'svc__C': 1, 'svc__gamma': 10}
Number of Support Vectors for Best Model: 197
F1-Score for Best Model: 0.9606
```

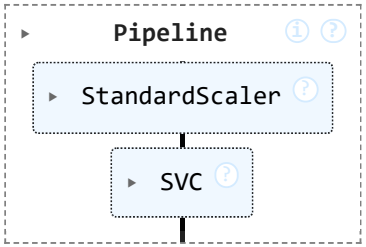
## 2.9

## Plot the data points and the decision boundary for the best RBF model.

We retrieve the best gamma and C values found during Grid Search and re-fit the SVM model using these parameters.

In [45]:
```
gamma = best_params['svc__gamma']
C = best_params['svc__C']

rbf_kernel_svm_clf = make_pipeline(StandardScaler(), SVC(kernel='rbf', gamma=gamma, C=C))
rbf_kernel_svm_clf.fit(X_selected, y)
```

Out[45]:




In [46]:
```
X_selected.shape
```

Out[46]:
```
(569, 2)
```

Since our model was trained on scaled data, we need to scale the data points for plotting to maintain consistency.

In [47]:
```
scaler = rbf_kernel_svm_clf.named_steps['standardscaler']
X_scaled_best = scaler.transform(X_selected)
```

We create the mesh grid for plotting. We start slightly below the minimum value, and we end slightly above the maximum value to ensure all data points are within the grid

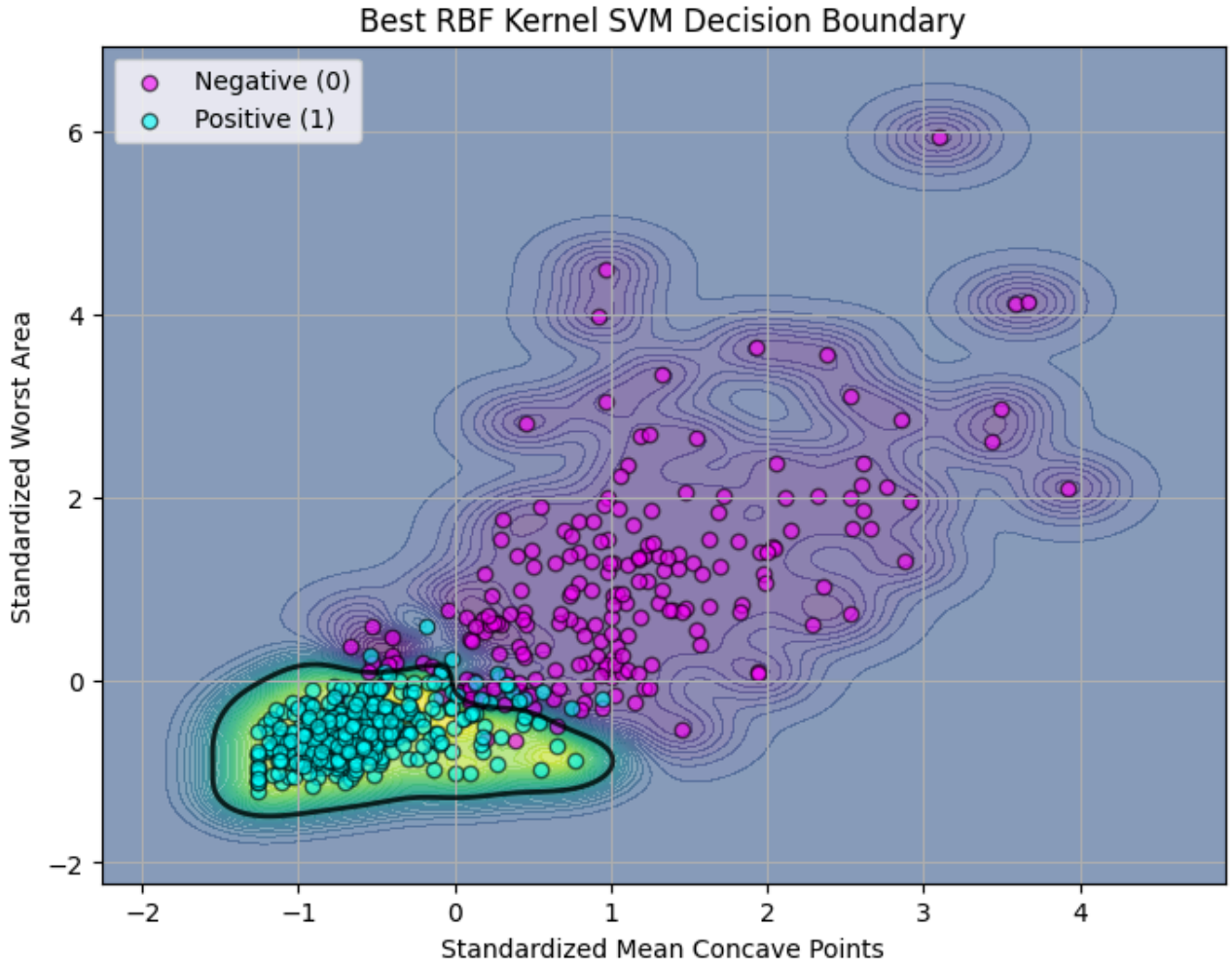
In [48]:
```
xx, yy = np.meshgrid(np.linspace(X_scaled_best[:, 0].min() - 1, X_scaled_best[:, 0].max() + 1, 200),
                     np.linspace(X_scaled_best[:, 1].min() - 1, X_scaled_best[:, 1].max() + 1, 200))
# xx and yy are (200,200) arrays. We flatten the 2D arrays into 1D and concatenate them to create a 2D array of grid points.
# Positive values indicate one class, negative values indicate the other.
grid = np.c_[xx.ravel(), yy.ravel()]
# The decision_function computes the distance of each point in grid to the decision boundary and creates an (40000,) array.
Z = rbf_kernel_svm_clf.named_steps['svc'].decision_function(grid)
# We, then reshape this array, to match the shape of xx and yy, so we can plot it over the grid.
Z = Z.reshape(xx.shape)
```

In [49]:
```
for name,var in {"xx":xx, "yy":yy, "Z":rbf_kernel_svm_clf.named_steps['svc'].decision_function(grid), "Z_Reshaped":Z}.items():
    print(f"Variable:{name}")
    print(type(var))
    print(var.shape)
```

```
Variable:xx
<class 'numpy.ndarray'>
(200, 200)
Variable:yy
<class 'numpy.ndarray'>
(200, 200)
Variable:Z
<class 'numpy.ndarray'>
(40000,)
Variable:Z_Reshaped
<class 'numpy.ndarray'>
(200, 200)
```

In [50]:

```python
# Plot decision boundary
plt.figure(figsize=(8, 6))
# Create filled contour plots
plt.contourf(xx, yy, Z, levels=np.linspace(Z.min(), Z.max(), 50), alpha=0.6, cmap='viridis')
# Draw the decision boundary where the decision function equals zero
plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='black', alpha=0.8)

# Plot the scaled data points
plt.scatter(X_scaled_best[y == 0, 0], X_scaled_best[y == 0, 1], c="magenta", label="Negative (0)", alpha=0.6, edgecolors="black")
plt.scatter(X_scaled_best[y == 1, 0], X_scaled_best[y == 1, 1], c="cyan", label="Positive (1)", alpha=0.6, edgecolors="black")

plt.xlabel("Standardized Mean Concave Points")
plt.ylabel("Standardized Worst Area")
plt.title("Best RBF Kernel SVM Decision Boundary")
plt.legend(loc='upper left')
plt.grid(True)
plt.show()
```




Unlike a linear SVM boundary, which was a single straight line, the RBF kernel produces a highly non-linear decision boundary. This allows the classifier to adapt to the data's underlying structure more flexibly, potentially capturing subtle class distinctions that a linear model misses.The best RBF model's decision boundary is a visually clear demonstration of how kernel methods can capture complex patterns in low-dimensional projections of high-dimensional data.