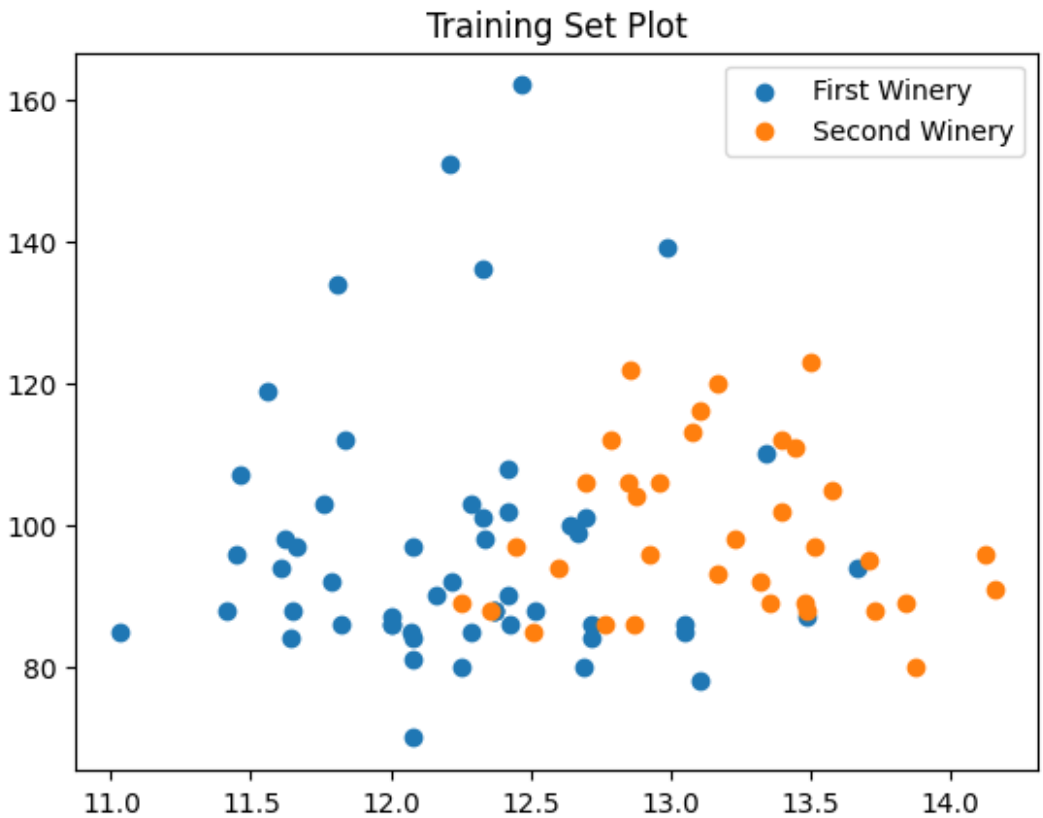
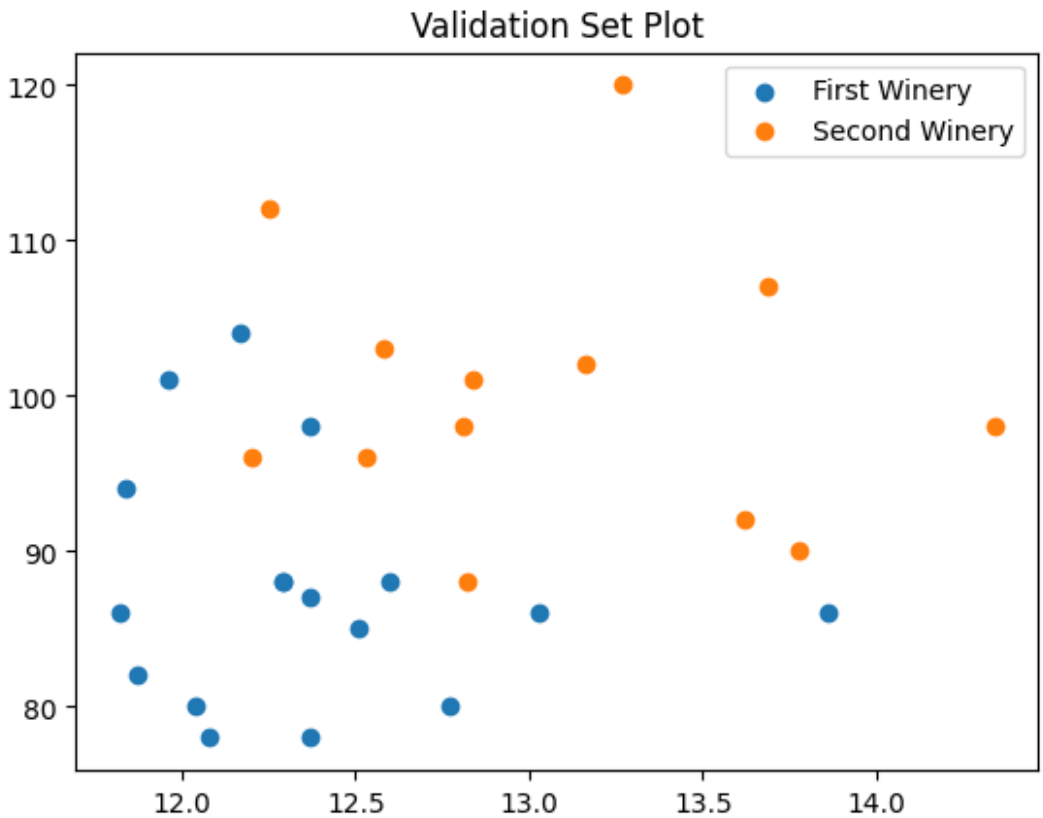
Question 1

Section 1.

```
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Read wine dataset
dataset = load_wine()
df = pd.DataFrame(data=dataset['data'],
columns=dataset['feature_names'])
df = df.assign(target=pd.Series(dataset['target']).values)

# Filter the irrelevant columns
df = df[['alcohol', 'magnesium', 'target']]
# Filter the irrelevant label
df = df[df.target != 0]

train_df, val_df = train_test_split(df, test_size=30, random_state=3)

def plot_wines(df, title, legend=True):
  # Plot wines DataFrame


  first = df[df.target == 1]
  second = df[df.target == 2]


  first_x = first['alcohol']
  first_y = first['magnesium']

  second_x = second['alcohol']
  second_y = second['magnesium']

  plt.scatter(first_x, first_y, label='First Winery')
  plt.scatter(second_x, second_y, label='Second Winery')
  plt.title(title)
  if legend:
    plt.legend()

plot_wines(train_df, 'Training Set Plot')
```




```
plot_wines(val_df, 'Validation Set Plot')
```

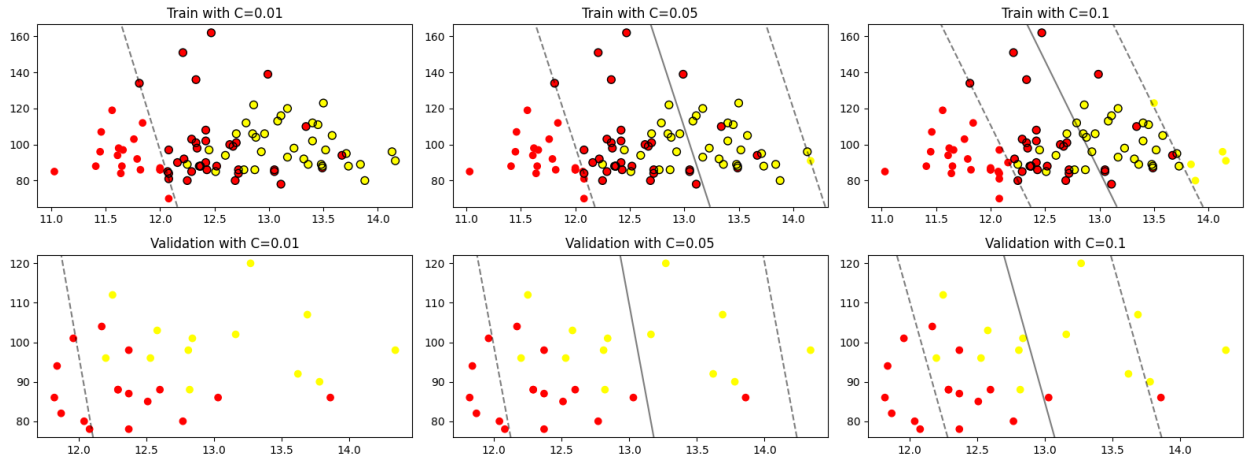


We can see that the data is not linearly separable. It follows that running hard-SVM will not give a result. The model will either run forever, return empty or throw an exception.

Section 2.

```
from sklearn.svm import SVC

def plot_svc_decision_function(model, ax=None, plot_support=True):
    """Plot the decision function for a 2D SVC"""
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # create grid to evaluate model
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)

    # plot decision boundary and margins
    ax.contour(X, Y, P, colors='k',
               levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])
```

```
    # plot support vectors
    if plot_support:
        ax.scatter(model.support_vectors_[:, 0],
                   model.support_vectors_[:, 1],
                   s=50, linewidth=1, facecolors='none',
edgecolor='black');
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)

X_train = train_df[['alcohol', 'magnesium']].to_numpy()
y_train = train_df['target'].to_numpy()

X_val = val_df[['alcohol', 'magnesium']].to_numpy()
y_val = val_df['target'].to_numpy()

fig, ax = plt.subplots(2, 3, figsize=(16, 6))

C_list = [0.01, 0.05, 0.1]

for i, C in enumerate(C_list):

  model = SVC(kernel='linear', C=C)
  model.fit(X_train, y_train)


  axi = ax[0, i]
  axi.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap='autumn')
  plot_svc_decision_function(model, axi)
  axi.set_title(f'Train with C={C}')

  axi = ax[1, i]
  axi.scatter(X_val[:, 0], X_val[:, 1], c=y_val, cmap='autumn')
  plot_svc_decision_function(model, axi, plot_support=False)
  axi.set_title(f'Validation with C={C}')

plt.tight_layout()
```



Section 3.

We need to prove that $min_{i \in [m]} |\langle \hat{w}, x_i \rangle| = \frac{1}{||w_0||}$.

Placing $\hat{w} = \frac{w_0}{||w_0||}$, we get:

$$min_{i \in [m]} |\langle \hat{w}, x_i \rangle| = min_{i \in [m]} \left| \frac{1}{||w_0||} \langle w_0, x_i \rangle \right| = \frac{1}{||w_0||} min_{i \in [m]} |\langle w_0, x_i \rangle|$$

By definition, support vectors are the vectors that are solve this optimization problem.

w_0 holds the following inequality:

$$y_i \langle w_0, x_i \rangle \geq 1 \Leftrightarrow |y_i \langle w_0, x_i \rangle| \geq 1 \Leftrightarrow |\langle w_0, x_i \rangle| \geq 1$$

Where the last inequality holds because $|y_i| = 1$. Therefore, we get:

$$min_{i \in [m]} |\langle \hat{w}, x_i \rangle| = \frac{1}{||w_0||} min_{i \in [m]} |\langle w_0, x_i \rangle| \geq \frac{1}{||w_0||}$$

Because $w_0 = argmin_w ||w||^2 : \forall i, y_i \langle w_0, x_i \rangle \geq 1$, and the support vectors are the ones that minimize this value, it holds that for any support vector $v : y_v \langle w_0, v \rangle = 1 \Leftrightarrow |y_v \langle w_0, v \rangle| = 1 \Leftrightarrow |\langle w_0, v \rangle| = 1$. Placing, we get:

$$\frac{1}{||w_0||} min_{i \in [m]} |\langle w_0, x_i \rangle| = \frac{1}{||w_0||} |\langle w_0, v \rangle| = \frac{1}{||w_0||}$$

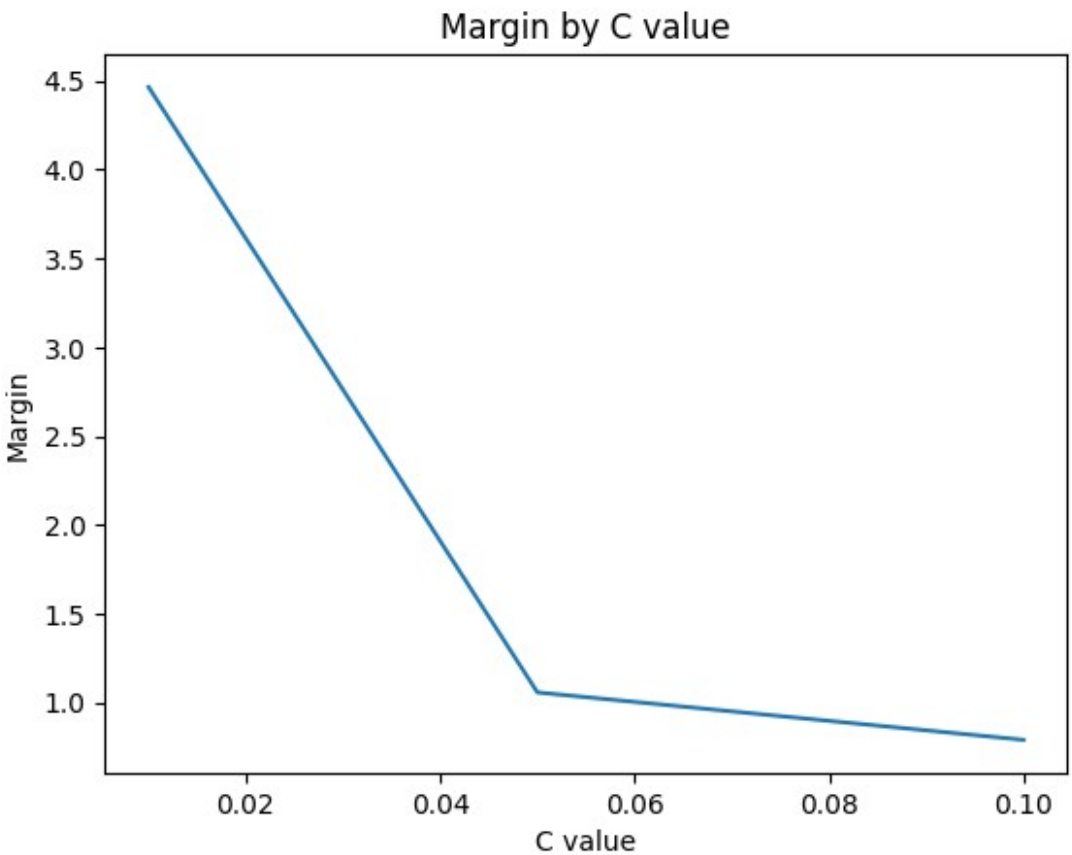
Concluding the proof.

Section 4.

```
y_values = []
```

```
for C in C_list:
  model = SVC(kernel='linear', C=C)
  model.fit(X_train, y_train)
  w_0 = model.coef_[0]

  margin = 1 / np.linalg.norm(w_0)
  y_values.append(margin)

plt.plot(C_list, y_values)
plt.xlabel('C value')
plt.ylabel('Margin')
plt.title('Margin by C value')
```

```
Text(0.5, 1.0, 'Margin by C value')
```

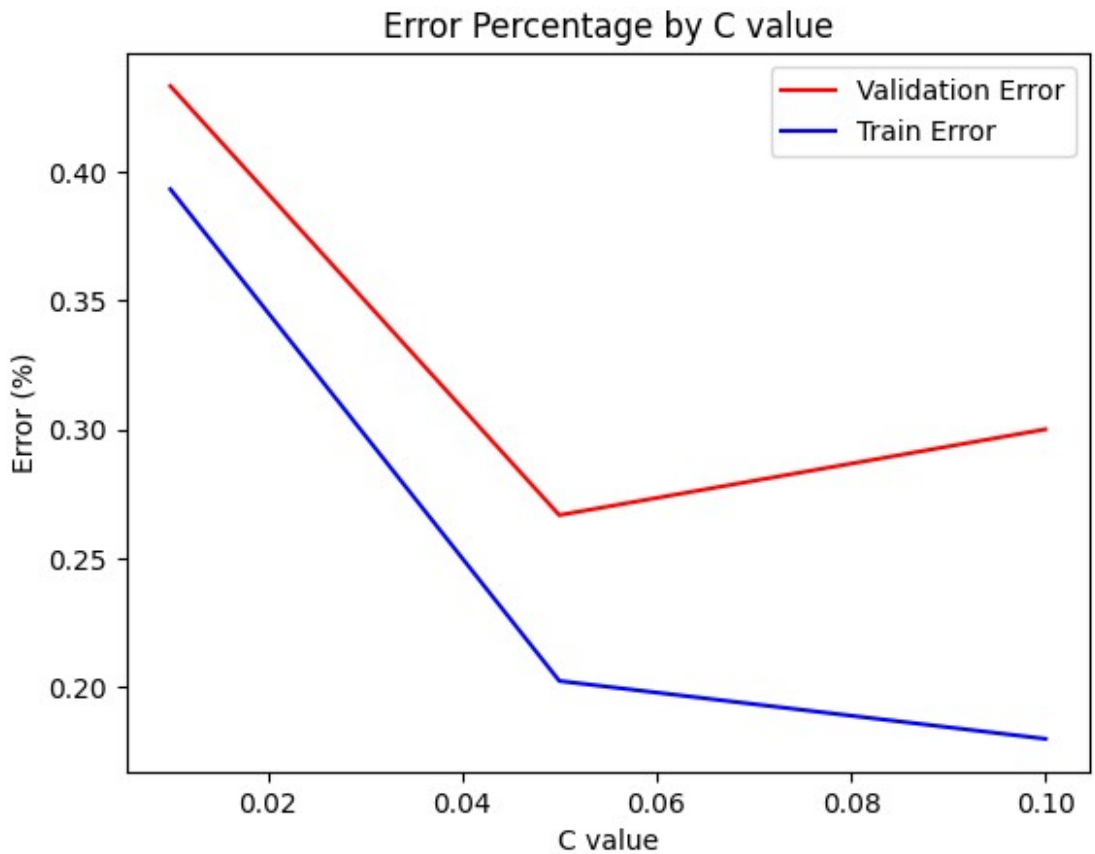


According to the graph, the higher C is, the worse the margin gets. This is consistent with what we learned in class: small C means we give more priority to a higher margin, while high C means giving higher priority to the test's accuracy at this stage (allowing for less loss).

Section 5.

```python
def calc_error(pred, actual):
  # Return the error percentage
  return np.sum(pred == actual) / len(pred)

train_y = []
val_y = []

for C in C_list:
  model = SVC(kernel='linear', C=C)
  model.fit(X_train, y_train)

  train_predict = model.predict(X_train)
  train_accuracy = calc_error(train_predict, y_train)

  val_predict = model.predict(X_val)
  val_accuracy = calc_error(val_predict, y_val)

  train_y.append(1-train_accuracy)
  val_y.append(1-val_accuracy)

plt.plot(C_list, val_y, c='red', label='Validation Error')
plt.plot(C_list, train_y, c='blue', label='Train Error')

plt.xlabel('C value')
plt.ylabel('Error (%)')
plt.legend()

plt.title('Error Percentage by C value')
```

```
Text(0.5, 1.0, 'Error Percentage by C value')
```



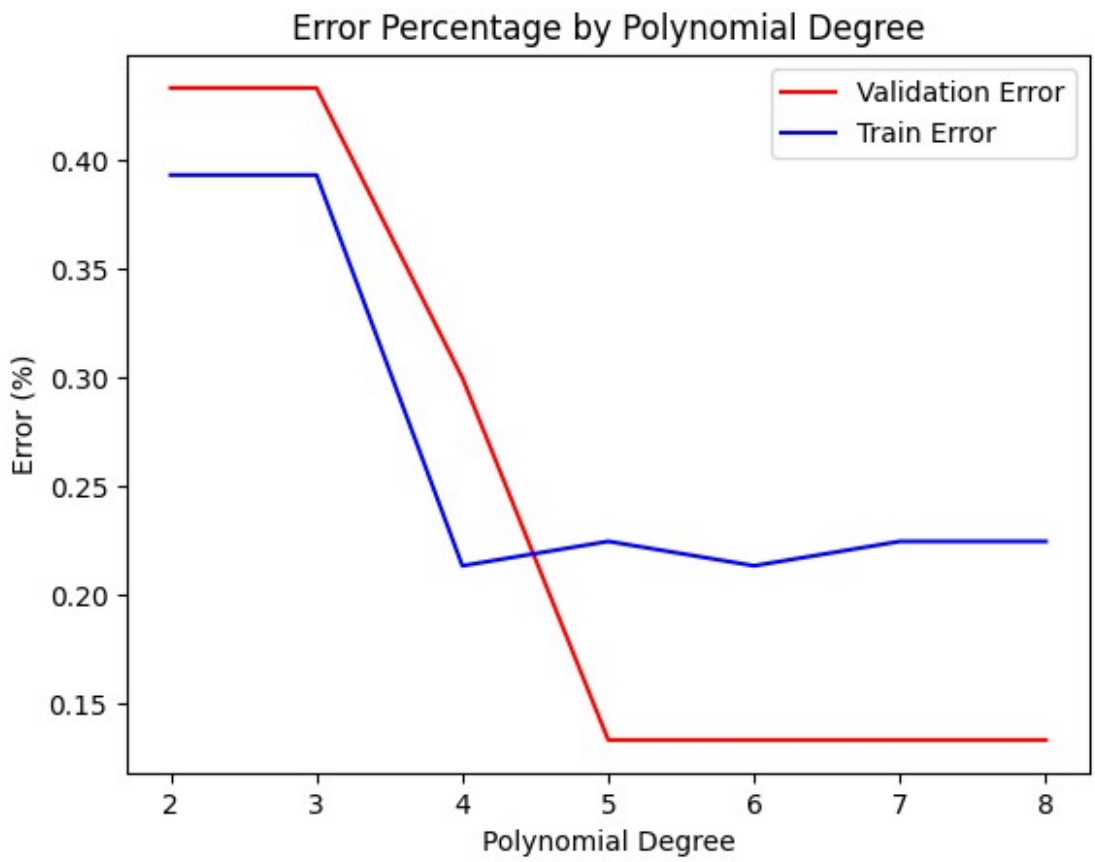
As we've said before: higher C value means giving more priority to being accurate on the training set. Thus, the graph's result is expected. We did bring the Train Error down by increasing C's value, but that came at a price of performing worse on the validation set.

Section 6.

```
C = 1
degree_list = range(2, 9)
train_y = []
val_y = []

for degree in degree_list:
  model = SVC(kernel='poly', C=C, degree=degree)
  model.fit(X_train, y_train)

  train_predict = model.predict(X_train)
  train_accuracy = calc_error(train_predict, y_train)

  val_predict = model.predict(X_val)
  val_accuracy = calc_error(val_predict, y_val)

  train_y.append(1-train_accuracy)
  val_y.append(1-val_accuracy)
```

```
plt.plot(degree_list, val_y, c='red', label='Validation Error')
plt.plot(degree_list, train_y, c='blue', label='Train Error')

plt.xlabel('Polynomial Degree')
plt.ylabel('Error (%)')
plt.legend()

plt.title('Error Percentage by Polynomial Degree')
```

```
Text(0.5, 1.0, 'Error Percentage by Polynomial Degree')
```



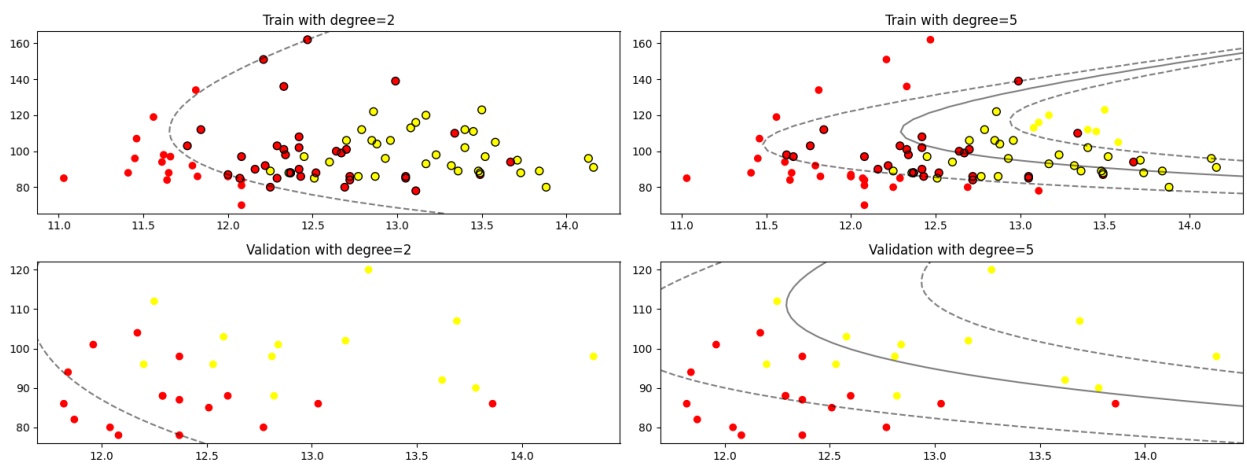
We can see that increasing the Polynomial's degree does not necessarily result in a better accuracy on the validation set. We can guess that this is because by increasing the degree too much, we might be giving weight to irrelevant dimensions which might make the model overfit or just perform worse in general.

Section 7.

The best and worst degrees are 2 and 5 (5 performs well on both validation and train).

```
fig, ax = plt.subplots(2, 2, figsize=(16, 6))
degree_list = [2, 5]
```

```
C = 1

for i, degree in enumerate(degree_list):

  model = SVC(kernel='poly', C=C, degree=degree)
  model.fit(X_train, y_train)


  axi = ax[0, i]
  axi.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap='autumn')
  plot_svc_decision_function(model, axi)
  axi.set_title(f'Train with degree={degree}')

  axi = ax[1, i]
  axi.scatter(X_val[:, 0], X_val[:, 1], c=y_val, cmap='autumn')
  plot_svc_decision_function(model, axi, plot_support=False)
  axi.set_title(f'Validation with degree={degree}')

plt.tight_layout()
```

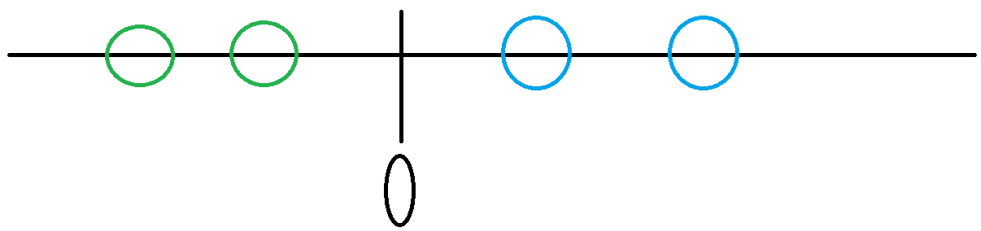


Question 3

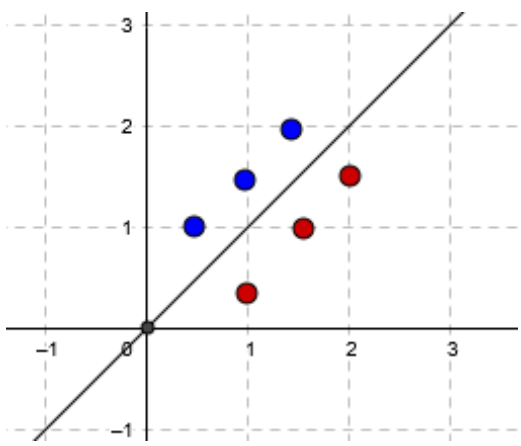
Section 1.

The classifier can get a training error rate of 0 if the data is linearly seperable, meaning if we can seperate between the 2 labels with a plane. Since we usually have a bias factor, which is 0 here, then the plane must go through the origin (0,0).

d = 1:



d=2:



Section 2.

$y_1\langle w, x_1\rangle = w_1 p, y_2\langle w, x_2\rangle = -w_2 q$

In order for a solution to exist, there must exist weights such that:

$w_1 p \geq 1, w_2 q \leq -1$

If either p or q are 0, then hard-SVM has no solution. Otherwise:

$w_1 \geq \frac{1}{p}, w_2 \leq -\frac{1}{q} \Rightarrow |)w|)^2 = w_1^2 + w_2^2 \geq \frac{1}{p^2} + \frac{1}{q^2}$

Therefore $w=\left(\frac{1}{p},\frac{1}{q}\right)$ is a solution to the quadtratic optimization formulation, which we proved is equivalent to hard-SVM.

Section 3.

Including the circled points would not change the separator, because the only points that require a change to the separator if changed are the support vectors, while all the circled points are outside the margin "tunnel".

Section 4.

If λ is huge like 200, then if the model makes some mistakes then it doesn't affect the outcome of soft-SVM dramatically. Therefore, Figure 1 fits this case, because the model classifies one of the points wrong, but in return has very big margins on both sides.

In contrast, for $\lambda=2$, every mistake the model makes matters a lot more. This would make the model more likely to overfit to the training set, while also causing it to have very small margins, which fits Figure 2.

So Figure 1 fits $\lambda=200$ while Figure 2 fits $\lambda=2$.

Question 4

Section 1.

This is what a "naive" standard perceptron using S^{ψ} would be:

input: A training set

$\{(\psi(x_1),y_1),\ldots,(\psi(x_m),y_m)\}, y_i \in \{-1,1\}$

initialize: $w^{(1)}=(0,\ldots,0)$

for $t=1,2,\ldots$

if $(\exists i \text{ s.t } y_i\langle w^{(t)},\psi(x_i)\rangle \leq 0)$ then

$w^{(t+1)}=w^{(t)}+y_i\psi(x_i)$

else

output $w^{(t)}$

This is obviously impossible to use as is, because we don't have ψ.

So, we want to reformulate this to fit the kernel function K.

We also note that to use the inner product of w with $\psi(x_i)$, $w \in F$ must hold, meaning w must be some combination of $\psi(x_i)$, where x_i are the training points.

We want $w^{(t)}$ to be a function of $\{\psi(x_k)\}_{i=1}^m$, so we'll define it as:

$$w^{(t)}=\sum_{i=1}^m \alpha_i \cdot \psi(x_i)$$

This fits the representer theorem seen in the tutorial, so we know that exist $\alpha_1,\ldots,\alpha_m \in R$ such that w solves the SVM optimization problem.

The decision function becomes

$$h(x)=\langle w,\psi(x)\rangle=\sum_{i=1}^m \alpha_i \langle \psi(x_i),\psi(x)\rangle=\sum_{i=1}^m \alpha_i K(x_i,x)$$

Or, more accurately,

$$h(x)=sign\left(\sum_{i=1}^m \alpha_i K(x_i,x)\right)$$

We can train the model by performing the following algorithm: For $i=1,2,\ldots$

If $\exists i: h(x_i)\neq y_i: \alpha_i=\alpha_i+1$;

We can't return w as the output directly because we don't have ψ, so we can instead return the parameters α_i, while also making sure we save the training vectors x_i.

Section 2.

$$w^{(t)}=\sum_{i=1}^m \alpha_i \psi(x_i)$$

We perform another step by checking the inner product, which we can calculate directly using K, the train labels and our parameters:

$$y_i \langle w^{(t)},x_i\rangle=y_i\sum_{j=1}^m \alpha_j K(x_j,x_i)$$

And note that

$$w^{(t+1)}=w^{(t)}+y_i\psi(x_i)=\sum_{j=1,\neq i}^m \alpha_j \psi(x_j)+(\alpha_i+1)y_i\psi(x_i)$$

So our algorithm is equivalent to using the "naive" perceptron pseudocode we suggested at the beginning, but with $S=S^\psi$, and the update rule is $\alpha_i \to \alpha_i+1$.

In the lecture, we proved that the perceptron algorithm only converges if and only if S is linearly separable, and since our algorithm is identical to the perceptron algorithm, then our algorithm also converges if and only if S^ψ is linearly separable.