Problem Statement

Datasets

You are provided with the training and testing datatset (see train.txt and test.txt), including 120 training data and 30 testing data, respectively. It covers 3 classes, corresponding to setosa, versicolor, virginica. They are derived from the Iris dataset (https://archive.ics.uci.edu/ml/datasets/ iris), contains 3 classes of 50 instances each, where each class refers to a type of iris plant. Your task is to classify each iris plant as one of the three possible types.

What you should do

You should use the SVM function from python sklearn package, which provides different form of SVM function you can use. For multiclass SVM you should use one vs rest strategy. You are recommended to use sklearn.svm.svc() function. You can use numpy for the vector manipulation. For technical report you should state clearly the optimization problem you are solving, how did you derive it, the meaning of different values in the formulation, and some results suitable for presenting in the report (e.g. training error, testing error). The basic form of SVM is given and you don't need to derive this

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2$$
$$\text{s.t. } 1 - y_i\left(\mathbf{w}^\top \mathbf{x}_i + b\right) \leq 0, \forall i$$

Problem 1 Calculate using standard SVM model (linear separator). Fit your algorithm on the training dataset, then validate your algorithm on testing dataset. Compute the misclassification error of training and testing datasets, the weight vector w, the bias b, and the indices of support vectors (start with 0). Write output to file SVM linear.txt. Note that the sklearn package doesn't provide a function with strict separation so we will simulate this using C = 1e5. You should print out the coefficient for each different class separately. The output format should be like this:

${training_error}
${testing_error}
${w_of_setosa}
${b_of_setosa}
${support_vector_indices_of_setosa}
${w_of_versicolor}
${b_of_versicolor}
${support_vector_indices_of_versicolor}
${w_of_virginica}
${b_of_virginica}
${support_vector_indices_of_virginica}

where each line contains one variable. The training error and testing error count the total error instead of error for each distinct class, the error is $\frac{\text{wrong prediction}}{\text{number of data}}$. If we view the one vs all strategy as combining the multiple different SVM, each one being a separating hyperplane for one class and the rest of the points, then the w, b and support vector indices for that class is the corresponding parameters for the SVM separating this class and the rest of the points. If a variable is of vector form, say $\alpha = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, then you should write each entry in the same line with comma separated, e.g.

```
1,2,3
```

You should also mention in your report on which classes are linear separable with SVM without slack and how you find it out.

2. Calculate using SVM with slack variables. For each $C = 0.1 \times t, t = 1, 2, ..., 10$, fit your algorithm on the training dataset, then validate your algorithm on testing dataset. Compute the misclassification error of training and testing datasets, the weight vector $\mathbf{w}$, the bias b, the indices of support vectors, and the slack variable $\boldsymbol{\xi}$. Write output to file **SVM_slack.txt**. The format is

```
${training_error}
${testing_error}
${w_of_setosa}
${b_of_setosa}
${support_vector_indices_of_setosa}
${slack_variable_of_setosa}
${w_of_versicolor}
${b_of_versicolor}
${support_vector_indices_of_versicolor}
${slack_variable_of_versicolor}
${w_of_virginica}
${b_of_virginica}
${support_vector_indices_of_virginica}
${slack_variable_of_virginica}
```

3. Implement SVM with kernel functions and slack variables. You should experiment with different kernel functions in this task:
   (a) A 2nd-order polynomial kernel, write output to **SVM_poly2.txt**
   (b) A 3rd-order polynomial kernel, write output to **SVM_poly3.txt**
   (c) Radial Basis Function kernel with $\sigma = 1$, write output to **SVM_rbf.txt**
   (d) Sigmoidal kernel with $\sigma = 1$, write output to **SVM_sigmoid.txt**

   During these tasks we set $C = 1$. The output format is

```
${training_error}
${testing_error}
${b_of_setosa}
${support_vector_indices_of_setosa}
${b_of_versicolor}
${support_vector_indices_of_versicolor}
${b_of_virginica}
${support_vector_indices_of_virginica}
```

Programming Problems

Problem Overview

For the following problems, the target is to do a **multi-class classification** over the Iris dataset provided (https://archive.ics.uci.edu/ml/datasets/iris) using the SVM model. The dataset covers 3 classes, corresponding to Sentosa, Versicolor, and Virginica. with 120 training data and 30 testing data respectively.

To implement multi-class SVM, the program uses the sklearn package and the **One vs Rest** strategy. Specifically, this strategy consists in fitting one classifier per class. For each classifier, the class is fitted against all the other classes. Therefore, only **n classes** classifiers are needed during the computation. This strategy is the most commonly used strategy for multiclass classification and is a fair default choice.

Problem 1: Linear SVM Classification

1.1 Mathematical model

The optimization problem corresponding to sklearn.svm.SVM() function (with kernel set to "linear") is given as

$$\min \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i}^{m}\xi_i$$

$$st. \quad 1-\xi_i - y_i(\boldsymbol{w}^T\boldsymbol{x_i}+b) \le 0, \qquad -\xi_i \le 0, \qquad \forall i$$

Since we manually set $\boldsymbol{C = 10^5}$, the model penalizes the slack variables to an extremely large extent so that it can be regarded as **a hard margin linear SVM model.**

Recall the prediction function is $y = \boldsymbol{w}^T\boldsymbol{x} + b$. As a result, if $\boldsymbol{w}^T\boldsymbol{x} + b > 0$, then the predicted class of $\boldsymbol{x}$ is +1, otherwise is -1

1.2 Implementation detail

The packages used for this problem are listed below:

```
import numpy as np
import pandas as pd
from sklearn.svm import SVC
from sklearn.multiclass import OneVsRestClassifier as OVR
```

Figure 3: Import Packages for Problem 1

This is the transformation function for y (labels) which the problem would use for conducting the one-vs-rest strategy

```
def transform_y(y_t, class_num):
    # print(class_num)
    y = np.array(y_t)
    y[y == class_num] = -1
    y[y != -1] = 1
    return y
```

Figure 4: transformation function

Next, the program does data preparation, which basically splits the x and y data and does the transformation for the y-labels.

```
# -------------data preparation-------------
df_train = pd.read_table("train.txt")
df_test = pd.read_table("test.txt")

data_train = df_train.to_numpy()
data_test = df_test.to_numpy()

x_train = np.delete(data_train, [0], axis=1)
y_train = data_train[:, 0]

x_test = np.delete(data_test, [0], axis=1)
y_test = data_test[:, 0]

y_train_Setosa = transform_y(y_train, 0)
y_train_Versicolour = transform_y(y_train, 1)
y_train_Virginica = transform_y(y_train, 2)

y_class = [y_train_Setosa, y_train_Versicolour, y_train_Virginica]
```

Figure 5: data preparation

The main logic for the SVM Modelling. Here are some interpretations given below:

```
# -------------SVM Modelling-------------
with open("SVM_linear.txt", 'w') as f:
    ovr_clf = OVR(SVC(C=1e5, kernel="linear")).fit(x_train, y_train)
    training_error = 1 - ovr_clf.score(x_train, y_train)
    testing_error = 1 - ovr_clf.score(x_test, y_test)

    f.write(str(training_error))
    f.write("\n")
    f.write(str(testing_error))
    f.write("\n")

    for i in y_class:
        clf = SVC(C=1e5, kernel="linear").fit(x_train, i)
        w = clf.coef_[0]
        b = clf.intercept_[0]
        svc_num = clf.n_support_[0]
        svid = clf.support_[:]

        w_ = ', '.join([str(i) for i in w])
        f.write(w_)
        f.write("\n")
        f.write(str(b))
        f.write("\n")
        id = ', '.join([str(i) for i in svid])
        f.write(id)
        f.write("\n")

f.close()
```

Figure 6: Main logic for SVM

1. For simplicity, the program uses the **OneVsRest** classifier with **.score()** method to calculate the training and testing error rather than taking the average for the n-class error which would basically produce the same results.
2. $\boldsymbol{w}$ and b in the prediction function are exported from the **classifier.coef_** and **classifier.intercepte_** attributes respectively.
3. The support vector indices are exported from the **classifier.support_** attribute; specifically, we can export the number of support vectors for each class from **classifier.n_support** attribute.

1.3 Analysis and Linear Separability

The total training and testing errors are given as:

```
0.04166666666666663
0.0
```

Figure 7: total error

The training and testing error for each class, following the sequence of Sentosa, Versicolor, and Virginica, during the one-vs-rest strategy are given as:

```
0.6666666666666667
0.6666666666666667


0.825
0.7666666666666666


0.675
0.6666666666666667
```

Figure 8: typical error for each class

```
0.0


0.22499999999999998


0.01666666666666672
```

Figure 9: Subtitute y_train with y_transformed

It is apparent that the first class **Sentosa is linearly separable to the rest of the two class** since it can be accurately classified into its own category which takes up 30% of the training and testing sample. (When substitute to transformed y, it becomes 0)

For the latter two classes, **Versicolor, and Virginica, they are not linearly separable** since they cannot be accurately classified into their own category when applying their one-vs-rest classifier to the training.

Problem 2: Linear SVM Classification with slack variables

2.1 Mathematical model

The corresponding optimization problem is given as

$$\min \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i}^{m}\xi_i$$

$$st. \quad 1-\xi_i - y_i(\boldsymbol{w}^T\boldsymbol{x_i}+b) \le 0, \qquad -\xi_i \le 0, \qquad \forall i$$

Since we manually set $C = 0.1 \times t, t = 1, 2, 3 \ldots 10$, the model penalizes the slack variables to a relatively small extent so that the model can tolerate some slackness.

The dual problem is given as:

$$\max \sum_{n=1}^{N}\alpha_i - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}\alpha_n\alpha_m y_n y_m \boldsymbol{x_n^T x_m}$$

$$st. \quad \sum_{i=1}^{N}\alpha_i y_i = 0, \qquad 0 \le \alpha_i \le C, \qquad \mu_i \ge 0, \qquad \alpha_i = C - \mu_i \qquad \forall i$$

KKT Condition for optimal situation:

$$\boldsymbol{w} = \sum_{i=1}^{N}\alpha_i y_i x_i\,, \qquad \sum_{i=1}^{N}\alpha_i y_i = 0$$

$$\alpha_i \ge 0, \qquad 1-\xi_i - y_i(\boldsymbol{w}^T\boldsymbol{x_i}+b) \le 0, \qquad \xi_i, \mu_i \ge 0 \qquad \forall i$$
$$\alpha_i\big(1-\xi_i - y_i(\boldsymbol{w}^T\boldsymbol{x_i}+b)\big) = 0, \qquad \xi_i\mu_i = 0 \qquad \forall i$$

Recall the prediction function is $y = \boldsymbol{w}^T\boldsymbol{x} + b$. As a result, if $\boldsymbol{w}^T\boldsymbol{x} + b > 0$, then the predicted class of $\boldsymbol{x}$ is +1, otherwise is -1

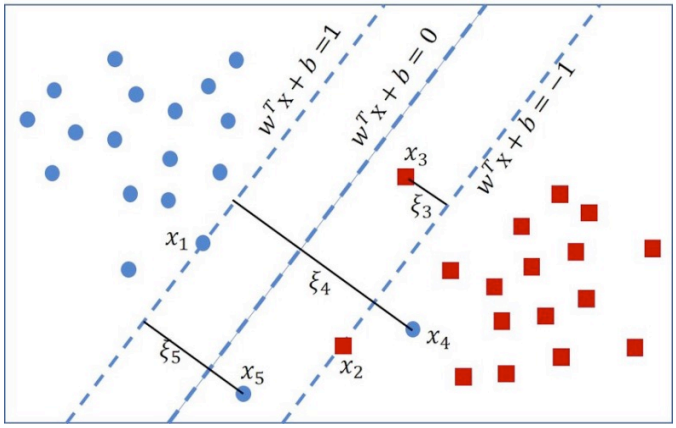

Figure 10: SVM with slack variable

To calculate the slack variable $\boldsymbol{\xi}$, we only need to consider the support vectors since other vector's slackness is regarded as 0:

$$\alpha_i = 0 \rightarrow \mu_i = C, \qquad \mu_i \xi_i = 0 \ (KKT) \rightarrow \xi_i = 0$$

Apparently, the support vector located on the hyperplane $\boldsymbol{w}^T\boldsymbol{x} + b = \pm 1$ should also have their slackness variable $\xi_i = 0$.

Hence, only need to consider the vector between $\boldsymbol{w}^T\boldsymbol{x} + b = 1$ and $\boldsymbol{w}^T\boldsymbol{x} + b = -1$.

2.2 Implementation detail

The problem structure basically follows the previous question, so here I only highlight some important differences.

```
def calc_slack(plane, svid):
    slack = np.zeros_like(plane)
    for i in svid:
        # round to 2-digit right to the decimal point
        slack[i] = np.around(1-plane[i], 2)
    return slack
```

Figure 11: calculation for slackness function

This function calculated the slack variable $\boldsymbol{\xi}$ based on the previous discussion in Section 2.1. The parameter **plane** is $y(\widehat{\boldsymbol{w}}^T\boldsymbol{x} + b)$, so theoretically all the value for this parameter should be within [0, 1]. Then, we only focus on the support vectors, with support vector indices provided in parameter **svid**, then calculate their ξ_i based on:

$$1 - \xi_i - y_i(\boldsymbol{w}^T\boldsymbol{x_i} + b) = 0 \ (KKT)$$

The results are also rounded to 2-digit decimal level.

2.3 Analysis outcome

The total training and testing errors are given as:

```
C = 0.1
0.125
0.23333333333333328

C = 0.2
0.05833333333333335
0.16666666666666663

C = 0.3
0.050000000000000044
0.13333333333333333

C = 0.4
0.050000000000000044
0.09999999999999998

C = 0.5
0.050000000000000044
0.09999999999999998

C = 0.6
0.050000000000000044
0.09999999999999998

C = 0.7
0.050000000000000044
0.09999999999999998

C = 0.8
0.050000000000000044
0.09999999999999998

C = 0.9
0.050000000000000044
0.06666666666666665

C = 1.0
0.050000000000000044
0.06666666666666665
```

Figure 12: Error with respect to C level

Problem 3: SVM Classification with kernel functions

3.1 Mathematical model

For different kernel functions, we can just set them in replace of the inner product in the SVM dual problem:

$$\boldsymbol{k}(\boldsymbol{x_n}, \boldsymbol{x_m}) = \phi(\boldsymbol{x_n})^T \phi(\boldsymbol{x_m})$$

$$\max \sum_{n=1}^{N} \alpha_i - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \phi(\boldsymbol{x_n})^T \phi(\boldsymbol{x_m})$$

$$st. \quad \sum_{i=1}^{N} \alpha_i y_i = 0, \qquad 0 \le \alpha_i \le C, \qquad \mu_i \ge 0, \qquad \alpha_i = C - \mu_i \qquad \forall i$$

Note that in this problem, hyper-parameter **C is fixed to 1**.

Correspondingly, the solution becomes:

$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i x_i \,, \qquad b = \frac{1}{|\mathcal{S}|} \sum_{j \in SV} (y_j - \sum_{i}^{N} \alpha_i y_i \, \boldsymbol{k}(\boldsymbol{x_i}, \boldsymbol{x_j}))$$

The kernel functions that the problem includes are given as

- linear: $\langle x, x' \rangle$.
- polynomial: $(\gamma \langle x, x' \rangle + r)^d$, where d is specified by parameter `degree`, r by `coef0`.
- rbf: $\exp(-\gamma \|x - x'\|^2)$, where γ is specified by parameter `gamma`, must be greater than 0.
- sigmoid $\tanh(\gamma \langle x, x' \rangle + r)$, where r is specified by `coef0`.

Figure 13: kernel functions in sklearn

3.2 Implementation detail

The problem structure basically follows the previous questions, so here I only highlight some important differences.

First, considering that in sigmoid functions y value is always within [0, 1], so in order to preserve consistence, the function is altered to transform y into {0, 1} for the sigmoid kernel case.

```
def transform_y(y_t, class_num, flag=0):
        # print(class_num)
        y = np.array(y_t)
        if flag == 0:
        # the target class we want to distinguish is set to -1
            y[y == class_num] = -1
            y[y != -1] = 1
            return y
        else:
            y[y == class_num] = 10
            y[y != 10] = 0
            y[y == 10] = 1
            return y
```

Figure 14: transform y (altered)

Another difference is that in the problem we **scale the data** to stand distribution for **the sigmoid kernel**. The reason is given below:

*"Standardization of datasets is a common requirement for many machine learning estimators implemented in scikit-learn; **they might behave badly** if the individual features do not more or less look like standard normally distributed data: **Gaussian with zero mean and unit variance.**"*

(Scikit learn 6.3 Document)

Specifically, for **sigmoid kernel**, if we don't scale the data to standard distribution, the training error and testing error will be **extremely high** (more than 90%). Therefore, we use the standard scaler to scale the data before fitting.

```
# scaling for x train dataset to standard distribution
scaler_1 = prep.StandardScaler().fit(x_train)
x_tr_scaled = scaler_1.transform(x_train)
scaler_2 = prep.StandardScaler().fit(x_test)
x_ts_scaled = scaler_2.transform(x_test)
```

Figure 15: scaling data

The last point. Since $\sigma = 1$ in this question, to set the **gamma** correspondingly, we can derive it from *Figure 13: kernel functions* that for **RBF gamma** should be **0.5**; However, this is **not applicable** for the **sigmoid**:

gamma : *{'scale', 'auto'} or float, default='scale'*

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if `gamma='scale'` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

Changed in version 0.22: The default value of `gamma` changed from 'auto' to 'scale'.

Figure 16: gamma description

And according the explanation given by the TA:

Q: The fourth question's sigmoid formula in the code doesn't match the one in the lecture slides

A: Indeed, the commonly used expression for the sigmoidal kernel is k(x, x_i) = tanh(\gamma x^Tx_i + r). We will set \gamma = 1/N, r = 0, where N is the dimension of x or x_i. If you set a different \gamma by yourself, please state it in your report.

Figure 17: TA Explanation

This problem use **gamma = 'auto'** in the sigmoid kernel function.

3.2 Analysis outcome

The total training and testing errors for different kernel functions are given as:

Polynomial:

```
Poly with degree 2
training error: 0.033333333333333326
testing error: 0.033333333333333326
Poly with degree 3
training error: 0.025000000000000022
testing error: 0.0
```

Figure 18: Polynomial kennel errors

RBF:

```
RBF Kernel
training error: 0.033333333333333326
testing error: 0.033333333333333326
```

Figure 19: RBF Kernel errors

Sigmoid:

```
Sigmoid Kernel
training error: 0.1166666666666667
testing error: 0.2666666666666667
```

Figure 20: Sigmoid kernel errors

Given the performances given above, we can conclude that polynomial kernel with degree 3 has the best classification performance in this question while the sigmoid kernel has the poorest performance.