COMP9417: A collection of sample exam exercises

August 9, 2022

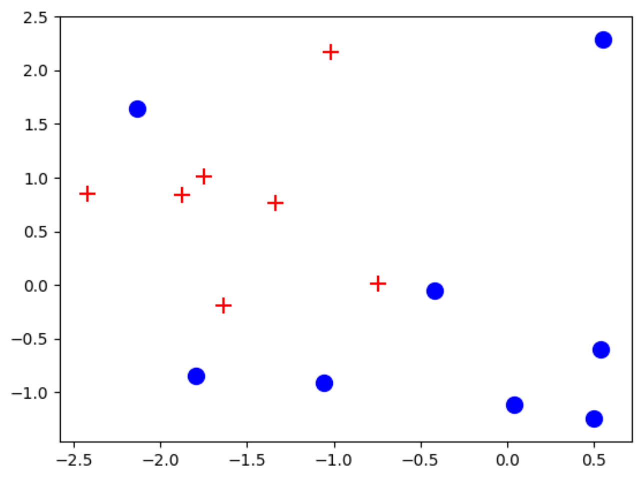
Note to student: Some of these questions are longer/more difficult than the questions that will be on the actual final exam. With that being said, you should aim to work through and understand all questions here as part of your revision. Note that this is not a sample final exam, it is a collection of possible exercises, and the actual final exam will be shorter.

Question 1

Note: this question was incorporated into the Ensemble learning lab and the code there is much improved. Refer to that lab for solutions. This question requires you to implement the Adaptive Boosting Algorithm from lectures. Use the following code to generate a toy binary classification dataset:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_blobs

np.random.seed(2)
n_points = 15
X, y = make_blobs(n_points, 2, centers=[(0,0), (-1,1)])
y[y==0] = -1       # use -1 for negative class instead of 0

plt.scatter(*X[y==1].T, marker="+", s=100, color="red")
plt.scatter(*X[y==-1].T, marker="o", s=100, color="blue")
plt.show()

```

Your data should look like:



(a) By now, you will be be familiar with the scikitlearn DecisionTreeClassifier class. Fit Decision trees of increasing maximum depth for depths ranging from 1 to 9. Plot the decision boundaries of each of your models in a 3×3 grid. You may find the following helper function useful:

```
def plotter(classifier, X, y, title, ax=None):
# plot decision boundary for given classifier
plot_step = 0.02
x_min, x_max = X[:, 0].min() - 1, X[:,0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:,1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                        np.arange(y_min, y_max, plot_step))
Z = classifier.predict(np.c_[xx.ravel(),yy.ravel()])
Z = Z.reshape(xx.shape)
if ax:
    ax.contourf(xx, yy, Z, cmap = plt.cm.Paired)
    ax.scatter(X[:, 0], X[:, 1], c = y)
    ax.set_title(title)
else:
    plt.contourf(xx, yy, Z, cmap = plt.cm.Paired)
    plt.scatter(X[:, 0], X[:, 1], c = y)
    plt.title(title)

```

(b) We now restrict attention to trees of depth 1. These are the most basic decision trees and are commonly referred to as decision stumps. Consider the adaptive boosting algorithm presented in the ensemble methods lecture notes on slide 50/70. In adaptive boosting, we build a model composed of T weak learners from a set of weak learners. At step t, we pick a model from the set of weak learners that minimises weighted error:

$$\epsilon_t = \sum_{i=1}^{n} w_{t-1,i} \mathbb{I}\{y_i \neq \hat{y}_i\}$$

where $w_{t-1,i}$ is the weight at the previous step for observation i, and $\mathbb{I}\{y_i \neq \hat{y}_i\}$ is equal to 1 if $y_i \neq \hat{y}_i$ and zero otherwise. We do this for a total of T steps, which gives us a boosted model composed of T base classifiers:

$$M(x) = \sum_{t=1}^{T} \alpha_t M_t(x)$$

where α_t is the weight assigned to the t-th model. Classification is then carried out by assigning a point to the positive class if $M(x) > 0$ or to the negative class if $M(x) < 1$. Here we will take the class of weak learners to be the class of Decision stumps. You may make use of the 'sample_weight' argument in the 'fit()' method to assign weights to the individual data points. Write code to build a boosted classifier for $T = 15$. Demonstrate the performance of your model on the generated dataset by printing out a list of your predictions versus the true class labels. (**note:** you may be concerned that the decision tree implementation in scikit learn does not actually minimise ϵ_t even when weights are assigned, but we will ignore this detail for the current question).

(c) In this question, we will extend our implementation in (c) to be able to use the plotter function in (b). To do this, we need to implement a boosting model class that has a 'predict' method. Once you do this, repeat (c) for $T = [2, \ldots, 17]$. Plot the decision boundary of your 16 models in a 4×4 grid. The following template may be useful:

```python
        class boosted_model:
            def __init__(self, T):
                self.alphas = # your code here
                # your code here

            def predict(self, x):
                # your code here

```

(d) Discuss the differences between bagging and boosting.

Question 2

Note: this question is actually a bit longer and (slightly) more difficult than an exam question, but it is good practice In this question, you will be required to manually implement Gradient Descent in Python to learn the parameters of a linear regression model. You will make use of the 'real_estate.csv' dataset, which you can download directly from the course Moodle page. The dataset contains 414 real estate records, each of which contains the following features:

- transactiondate: date of transaction
- age: age of property
- nearestMRT: distance of property to nearest supermarket
- nConvenience: number of convenience stores in nearby locations
- latitude
- longitude

The target variable is the property price. The goal is to learn to predict property prices as a function of a subset of the above features.

(a) A number of pre-processing steps are required before we attempt to fit any models to the data. First remove any rows of the data that contain a missing ('NA') value. List the indices of the removed data points. Then, delete all features from the dataset apart from: age, nearestMRT and nConvenience.

Solution:

The indices of rows with missing values are: $[19, 41, 109, 144, 230, 301]$.

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
df = pd.read_csv("real_estate.csv")
df = df.dropna()
X = df[["age", "nearestMRT", "nConvenience"]].values

```

(b) An important pre-processing step: feature normalisation. Feature normalisation involves rescaling the features such that thay all have similar scales. This is also important for algorithms like gradient descent to ensure the convergence of the algorithm. One common technique is called min-max normalisation, in which each feature is scaled to the range $[0, 1]$. To do this, for each feature we must find the minimum and maximum values in the available sample, and then use the following formula to make the transformation:

$$x_{\text{new}} = \frac{x - \min(x)}{\max(x) - \min(x)}.$$

After applying this normalisation, the minimum value of your feature will be 0, and the maximum will be 1. For each of the features, provide the mean value over your dataset.

Solution:

The mean values for the three features are: $[0.40607933, 0.16264268, 0.4120098]$.

```
# preprocessing
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
y = df["price"].values
```

(c) Now that the data is pre-processed, we will create train and test sets to build and then evaluate our models on. Use the first half of observations (in the same order as in the original csv file excluding those you removed in the previous question) to create the training set, and the remaining half for the test set. Print out the first and last rows of both your training and test sets.

Solution:

- first row Xtrain: $[0.73059361, 0.00951267, 1.]$
- last row Xtrain: $[0.87899543, 0.09926012, 0.3]$
- first row Xtest: $[0.26255708, 0.20677973, 0.1]$
- last row Xtest: $[0.14840183, 0.0103754, 0.9]$
- first row ytrain: 37.9
- last row ytrain: 34.2
- first row ytest: 26.2
- last row ytest: 63.9

```
# train/test splot
X = np.concatenate((np.ones([len(X), 1]), X), axis=1)
split_point = X.shape[0] // 2
X_train = X[:split_point]
X_test = X[split_point:]
y_train = y[:split_point]
y_test = y[split_point:]

# printing
print("first row Xtrain: ", Xtrain[0])
print("last row Xtrain: ", Xtrain[-1])
print("first row Xtest: ", Xtest[0])
print("last row Xtest: ", Xtest[-1])
print("first row ytrain: ", ytrain[0])
print("last row ytrain: ", ytrain[-1])
print("first row ytest: ", ytest[0])
print("last row ytest: ", ytest[-1])
```

(d) Consider the loss function

$$\mathcal{L}_c(x, y) = \sqrt{\frac{1}{c^2}(x-y)^2 + 1} - 1,$$

where $c \in \mathbb{R}$ is a hyper-parameter. Consider the (simple) linear model

$$\hat{y}^{(i)} = w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} + w_3 x_3^{(i)}, \qquad i = 1, \dots, n.$$

We can write this more succintly by letting $w = (w_0, w_1, w_2, w_3)^T$ and $X^{(i)} = (1, x_1^{(i)}, x_2^{(i)}, x_3^{(i)})^T$, so that $\hat{y}^{(i)} = w^T X^{(i)}$. The mean-loss achieved by our model (w) on a given dataset of n observations is then

$$\mathcal{L}_c(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_c(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{n} \sum_{i=1}^{n} \left[\sqrt{\frac{1}{c^2} (y^{(i)} - \langle w^{(t)}, X^{(i)} \rangle)^2 + 1} - 1 \right],$$

Compute the following derivatives:

$$\frac{\partial \mathcal{L}_c(y^{(i)}, \hat{y}^{(i)})}{\partial w_k}, \qquad k = 0, 1, 2, 3.$$

You must show your working for full marks.

Solution:
After applying the chain rule twice, we get results similar to the L2 loss cases:

$$\frac{\partial \mathcal{L}_c(y^{(i)}, \hat{y}^{(i)})}{\partial w_k} = \frac{X_k^{(i)} (\langle w^{(t)}, X^{(i)} \rangle - y^{(i)})}{c^2 \sqrt{\frac{1}{c^2} (\langle w^{(t)}, X^{(i)} \rangle - y^{(i)})^2 + 1}}, \qquad k = 0, 1, 2, 3.$$

(e) For some value of $c > 0$, we have

$$\frac{\partial \mathcal{L}_c(y^{(i)}, \hat{y}^{(i)})}{\partial w_k} = \frac{x_k^{(i)} (\langle w^{(t)}, X^{(i)} \rangle - y^{(i)})}{2\sqrt{(\langle w^{(t)}, X^{(i)} \rangle - y^{(i)})^2 + 4}}, \qquad k = 0, 1, 2, 3.$$

Using this result, write down the gradient descent updates for w_0, w_1, w_2, w_3 (using pseudocode), assuming a step size of η. Note that in gradient descent we consider the loss over the entire dataset, not just at a single observation. Further, provide pseudocode for stochastic gradient descent updates. Here, $w^{(t)}$ denotes the weight vector at the t-th iteration of gradient descent.

Solution:
We have the following updates for GD, for $t = 1, \dots, T$:

$$w_k^{(t+1)} = w_k^{(t)} - \eta \frac{1}{n} \sum_{i=1}^{n} \frac{x_k^{(i)} (\langle w^{(t)}, X^{(i)} \rangle - y_i)}{2\sqrt{(\langle w^{(t)}, X^{(i)} \rangle - y_i)^2 + 4}}$$

Similarly for SGD, we have updates

$$w_k^{(t+1)} = w_k^{(t)} - \eta \frac{x_k^{(i)} (\langle w^{(t)}, X^{(i)} \rangle - y_i)}{2\sqrt{(\langle w^{(t)}, X^{(i)} \rangle - y_i)^2 + 4}}$$

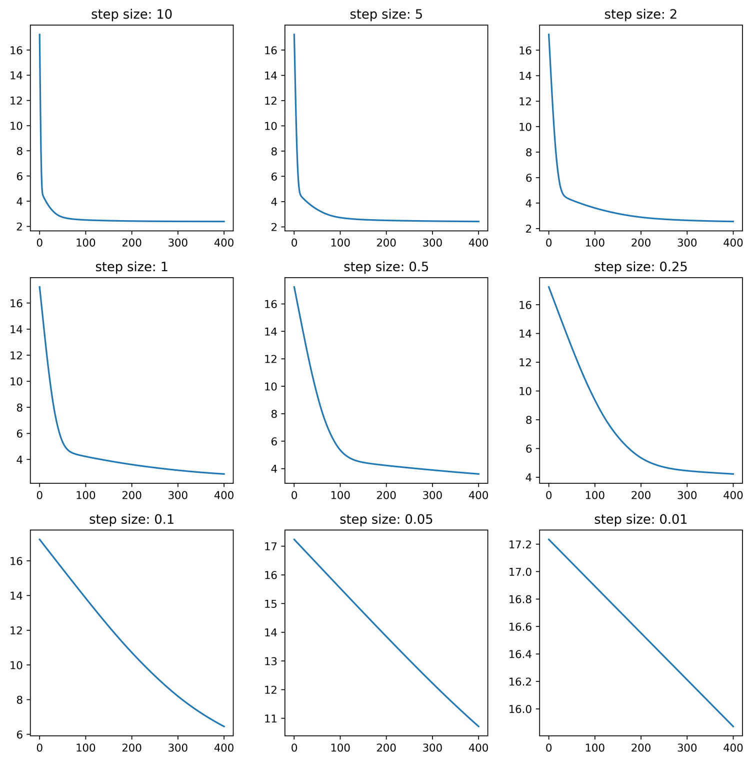
(f) In this section, you will implement gradient descent from scratch on the generated dataset using the gradients computed in Question 3, and the pseudocode in Question 4. Initialise your weight vector to $w^{(0)} = [1, 1, 1, 1]^T$. Consider step sizes

$$\eta \in \{10, 5, 2, 1, 0.5, 0.25, 0.1, 0.05, 0.01\}$$

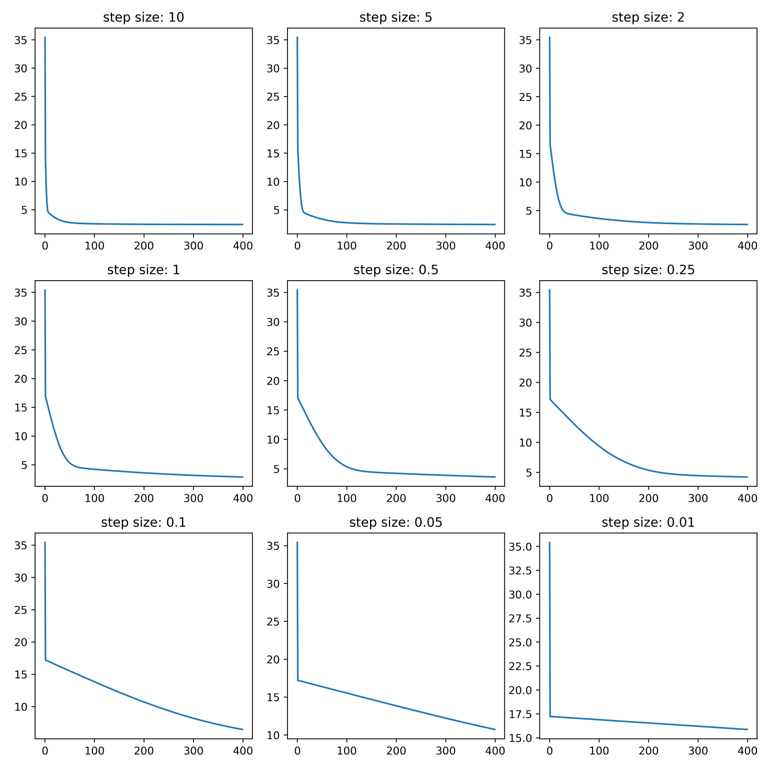
(a total of 9 step sizes). For each step-size, generate a plot of the loss achieved at each iteration of gradient descent. You should use 400 iterations in total (which will require you to loop over your training data in order). Generate a 3×3 grid of plots showing performance for each step-size. Add a screen shot of the python code written for this question in your report (as well as pasting the code in to your solutions.py file). The following code may help with plotting:

```python
            fig, ax = plt.subplots(3,3, figsize=(10,10))
            nIter = 400
            alphas = [10,5,2, 1,0.5, 0.25,0.1, 0.05, 0.01]
            for i, ax in enumerate(ax.flat):
                # losses is a list of 9 elements. Each element is an array of length
    nIter storing the loss at each iteration for
                # that particular step size
                ax.plot(losses[i])
                ax.set_title(f"step size: {alphas[i]}")  # plot titles
            plt.tight_layout()      # plot formatting
            plt.show()

```

Solution:
The plot should look like:



If the student did not figure out that the provided gradient corresponds to the case $c = 2$, then they were told they could use either $c = 1$ or $c = 2$. This affects their loss values, even though they are using the correct gradients, and in that case their plots would look like:



Example code is as follows:

```python
def calc_loss(y, y_pred, c=2):
    return np.mean(np.sqrt((1/c**2) * (y - y_pred)**2 + 1) - 1)

def calc_grad(X_train, y_train, w, c=2):
    Xw = X_train @ w
    const = (Xw - y_train) / (c**2 * np.sqrt((1/c**2) * (Xw - y_train)**2 + 1))
    grad = np.mean(X_train * np.repeat(const, 4).reshape(-1, 4), axis=0)
    return grad

nmb_iter = 400
w = np.zeros(shape=(nmb_iter+1, 4))
w[0] = np.array([1,1,1,1])

alphas = [10,5,2,1,0.5,0.25,0.1,0.05, 0.01]
losses = []
cur_c = 2
for alpha in alphas:
    train_loss = np.zeros(shape=(nmb_iter+1))
    train_loss[0] = calc_loss(ytrain, Xtrain @ w[0], c=cur_c)
    for i in range(1, nmb_iter+1):
        w[i] = w[i-1] - alpha * calc_grad(Xtrain, ytrain, w[i-1])
        train_loss[i] = calc_loss(ytrain, Xtrain @ w[i], c=cur_c)
    losses.append(train_loss)
```

```
            fig, ax = plt.subplots(3,3, figsize=(10,10))
            for i, ax in enumerate(ax.flat):
                ax.plot(losses[i])
                ax.set_title(f"step size: {alphas[i]}")

            plt.tight_layout()
            plt.savefig("GDgrid.png", dpi=500)
            plt.show()
```

(g) From your results in the previous part, choose an appropriate step size (and state your choice), and explain why you made this choice.

Solution:
$\eta = 10$ seems like the best choice.

(h) For this part, take $\eta = 0.3$, re-run GD under the same parameter initilisations and number of iterations. On a single plot, plot the progression of each of the four weights over the iterations. Print out the final weight vector. Finally, run your model on the train and test set, and print the achieved losses.

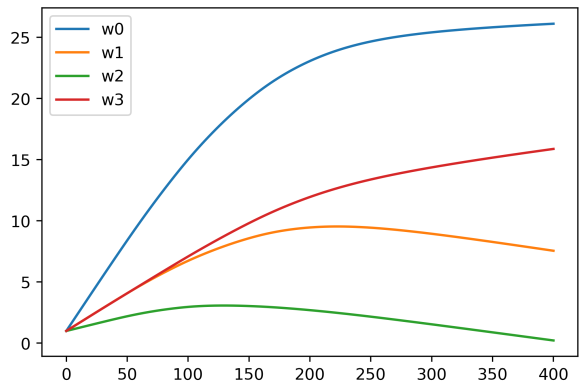
Solution:
We have

$$w = [26.11838076, 7.55495429, 0.22091267, 15.88216107]$$

Rerunning this model gives the following results:

- train loss: 4.089850739201336 / 8.887715476132039 (if $c = 1$)
- test loss: 3.8275009292188362 / 8.351707785988028 (if $c = 1$)

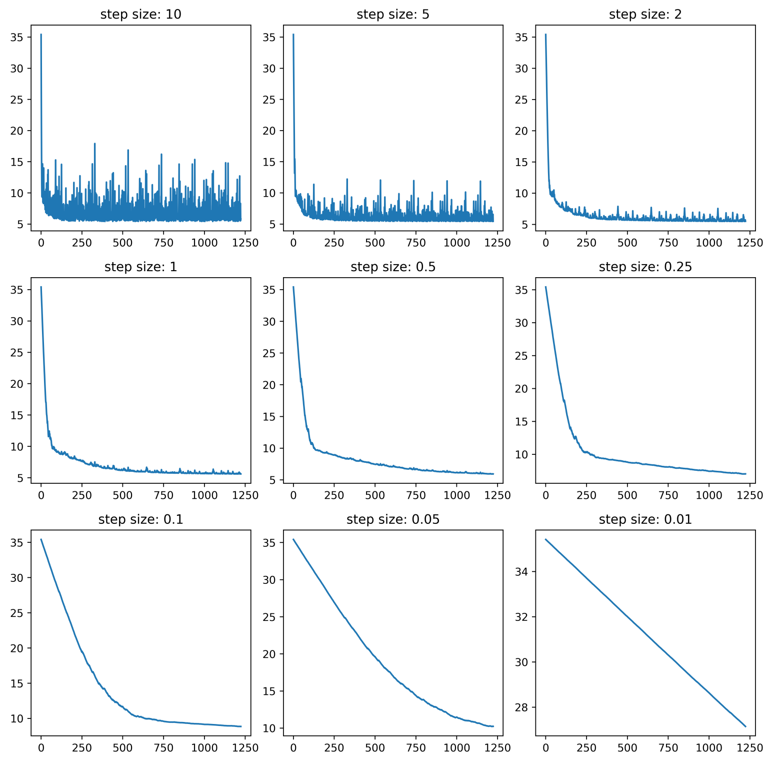
The plot of the weights look like:



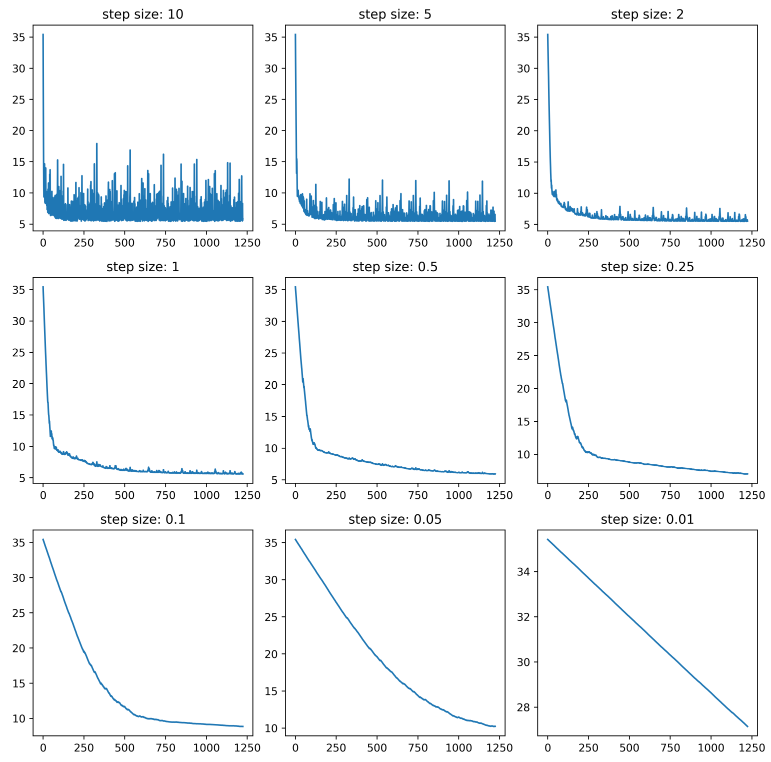
```python
w = np.zeros(shape=(nmb_iter,4))
w[0] = np.array([1,1,1,1])

alpha = 0.3
for i in range(1, nmb_iter):
    w[i] = w[i-1] - alpha * calc_grad(Xtrain, ytrain, w[i-1])

plt.plot(w[:,0], label="w0")
plt.plot(w[:,1], label="w1")
plt.plot(w[:,2], label="w2")
plt.plot(w[:,3], label="w3")
plt.legend()
plt.savefig("GDweights.png", dpi=400)
plt.show()

wstar = w[nmb_iter-1]
print("train loss: ", calc_loss(ytrain, Xtrain @ wstar, c=cur_c))
print("test loss: ", calc_loss(ytest, Xtest @ wstar, c=cur_c))
```

(i) We will now re-run the analysis in the previous part, but using stochastic gradient descent (SGD) instead. In SGD, we update the weights after seeing a single observation. Use the same settings as in the gradient descent implementation. For each step-size, generate a plot of the loss achieved at each iteration of stochastic gradient descent, which is to be run for 6 Epochs. An epoch is one pass over the entire training dataset, so in total there should be $6 \times n_{\text{train}}$ updates, where n_{train} is the size of your training data. **Be sure to run these updates in the same ordering that the data is stored.** Generate a 3×3 grid of plots showing performance for each step-size. Add a screen shot of the python code written for this question in your report (as well as pasting the code in to your solutions.py file).

Solution:
The plot should look like:



If students use $c = 1$, then



Example code is as follows:

```
            def calc_grad_i(w, Xi, yi, c=2):
                # compute gradient for a single observation
                xw = Xi@w
                num = xw-yi
                den = (c**2) * np.sqrt(1 + (1/c**2) * (xw-yi)**2)

                return (num/den)*Xi

            epochs = 6
            nmb_iter = Xtrain.shape[0] * epochs
            w = np.zeros(shape=(nmb_iter,4))
            w[0] = np.array([1,1,1,1])

            alphas = [10, 5, 2, 1, 0.5, 0.25, 0.1, 0.05, 0.01]
            losses = []
            cur_c = 2
            for alpha in alphas:
                train_loss = np.zeros(shape=(nmb_iter))
                train_loss[0] = calc_loss(ytrain, Xtrain @ w[0], c=cur_c)
                for i in range(1, nmb_iter):
                    idx = i % Xtrain.shape[0]
                    w[i] = w[i-1] - alpha * calc_grad_i(w[i-1], Xtrain[idx], ytrain[
    idx], c=cur_c)
                    train_loss[i] = calc_loss(ytrain, Xtrain @ w[i], c=cur_c)
                losses.append(train_loss)
```

```
        fig, ax = plt.subplots(3,3, figsize=(10,10))
        for i, ax in enumerate(ax.flat):
            ax.plot(losses[i])
            ax.set_title(f"step size: {alphas[i]}")

        plt.tight_layout()
        plt.savefig("SGDgrid.png", dpi=500)
        plt.show()
```

(j) From your results in the previous part, choose an appropriate step size (and state your choice), and explain why you made this choice.

Solution:
0.5 seems like a reasonable choice, but 0.25 is also acceptable.

(k) Take $\eta = 0.4$ and re-run SGD under the same parameter initilisations and number of epochs. On a single plot, plot the progression of each of the four weights over the iterations. Print yoru final model. Finally, run your model on the train and test set, and record the achieved losses.

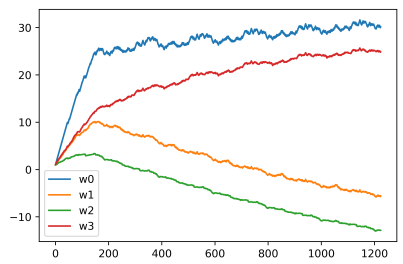
Solution:
We have

$$w = [30.13933686, -5.58250367, -12.85536393, 24.85002152]$$

Rerunning this model gives the following results:

- train loss: 2.7805130067939547 / 6.181134362027168 (if $c = 1$)
- test loss: 2.8446116656442526 / 6.329857966449784 (if $c = 1$)

The plot of the weights look like:



```
            w = np.zeros(shape=(nmb_iter,4))
            w[0] = np.array([1,1,1,1])
```

```
                alpha = .4
                loss_vec = np.zeros(shape=(nmb_iter))
                loss_vec[0] = calc_loss(ytrain, Xtrain @ w[0], c=cur_c)
                for i in range(1, nmb_iter):
                    idx = i % Xtrain.shape[0]
                    w[i] = w[i-1] - alpha * calc_grad_i(w[i-1], Xtrain[idx], ytrain[
    idx], c=cur_c)
                    loss_vec[i] = calc_loss(ytrain, Xtrain @ w[i], c=cur_c)

                plt.plot(w[:,0], label="w0")
                plt.plot(w[:,1], label="w1")
                plt.plot(w[:,2], label="w2")
                plt.plot(w[:,3], label="w3")
                plt.legend()
                plt.savefig("SGDweights.png", dpi=500)
                plt.show()

                wstar = w[-1]

                print("w: ", wstar)
                print("train loss: ", calc_loss(Xtrain@wstar, ytrain, c=cur_c))
                print("test loss: ", calc_loss(Xtest@wstar, ytest, c=cur_c))

```

(l) In a few lines, comment on your results in Questions 5 and 6. Explain the importance of the step-size in both GD and SGD. Explain why one might have a preference for GD or SGD. Explain why the GD paths look much smoother than the SGD paths.

Solution:
This is an open question so any reasonable short answer should be given full marks. Students should mention things along the lines of:

- In general, Gradient descent takes steps towards the minimum by moving in the direction of greatest descent. For large step sizes, we may overshoot the minimum and this may result in oscillating or divergence of the algorithm, whereas too small a stepsize will lead to very slow convergence. We will in general find that GD uses a larger step size relative to SGD, since the direction (gradient) is more reliable in GD than in SGD.
- SGD is an approximation of GD, since in GD we use the entire training data before making a single update, whereas in SGD we make updates after seeing a single training point. The reason we might prefer SGD is when it is too computationally expensive to compute the gradient over the entire dataset, and SGD tends to converge much faster than GD for this reason. We see that in this case, SGD is able to achieve a lower mean loss, though we cannot directly compare the two algorithms here as the number of iterations differs, it would be interesting if a student provides a timing comparison for the two cases but not necessary.
- The SGD paths are much more turbulent because we are estimating the gradient by looking at a single training point, and so the gradient updates are much noisier than when computing the gradient over the entire training set as in standard GD.
- Any other discussion points, or mention of mini-batch GD, or other variants.

Question 3

In this question, we will consider the scikitlearn implementations of the following classifiers:

- Decision Trees
- Random Forest
- AdaBoost
- Logistic Regression
- Multilayer Perceptron (Neural Network)
- Support Vector Machine

You are required to compare the performance of the above models on a classifiation task. The following code loads in these classifiers and defines a function to simulate a toy dataset:

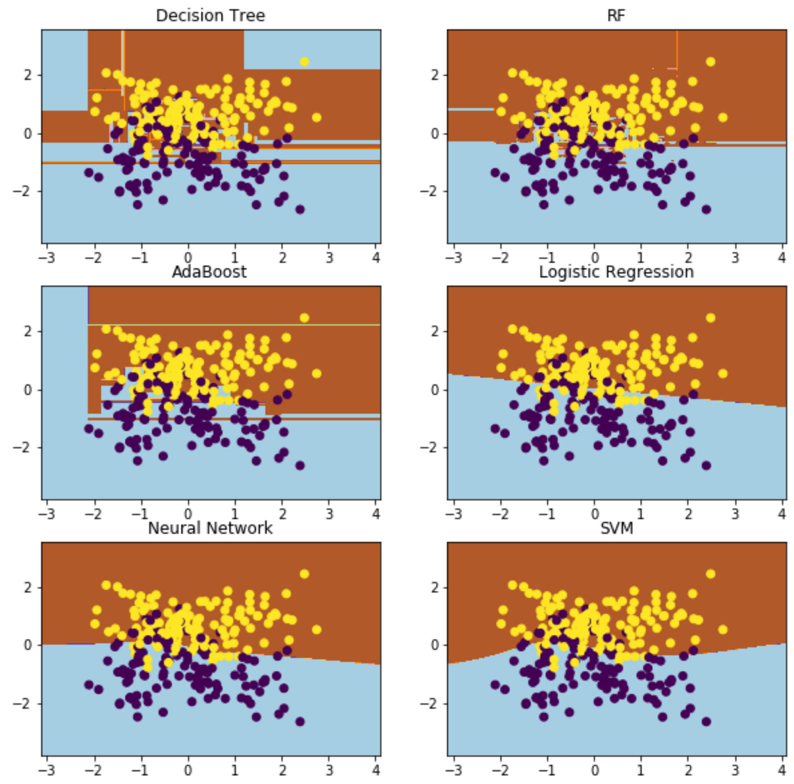
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

import time
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification

def create_dataset(n=1250, nf=2, nr=0, ni=2, random_state=125):
    '''
    generate a new dataset with
    n: total number of samples
    nf: number of features
    nr: number of redundant features (these are linear combinatins of
informative features)
    ni: number of informative features (ni + nr = nf must hold)
    random_state: set for reproducibility
    '''
    X, y = make_classification( n_samples=n,
                                n_features=nf,
                                n_redundant=nr,
                                n_informative=ni,
                                random_state=random_state,
                                n_clusters_per_class=2)
    rng = np.random.RandomState(2)
    X += 3 * rng.uniform(size = X.shape)
    X = StandardScaler().fit_transform(X)
    return X, y
```

(a) Generate a dataset using the default parameters of `create_dataset`. Then, randomly split the dataset into training set Xtrain and test set Xtest (use the sklearn `train_test_split` function with `random_state=45`), with 80 % of examples for training and 20 % for testing. Fit each of the models to the training data and then plot the decision boundaries of each of the classifiers (using default parameter settings) on the test set. If you prefer, you may use the following plotter function which plots the decision boundary and works for any sklearn model.

```
            def plotter(classifier, X, X_test, y_test, title, ax=None):
                # plot decision boundary for given classifier
                plot_step = 0.02
                x_min, x_max = X[:, 0].min() - 1, X[:,0].max() + 1
                y_min, y_max = X[:, 1].min() - 1, X[:,1].max() + 1
                xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                                        np.arange(y_min, y_max, plot_step))
                Z = classifier.predict(np.c_[xx.ravel(),yy.ravel()])
                Z = Z.reshape(xx.shape)
                if ax:
                    ax.contourf(xx, yy, Z, cmap = plt.cm.Paired)
                    ax.scatter(X_test[:, 0], X_test[:, 1], c = y_test)
                    ax.set_title(title)
                else:
                    plt.contourf(xx, yy, Z, cmap = plt.cm.Paired)
                    plt.scatter(X_test[:, 0], X_test[:, 1], c = y_test)
                    plt.title(title)

```

Note that you can use the same general approach for plotting a grid as used in Homework 1, and the plotter function supports an 'ax' argument. *What to submit: a single* 3×2 *plot, a print screen of the python code used, a copy of your python code in solutions.py.*

Solution:
The students should generate a plot that looks like:



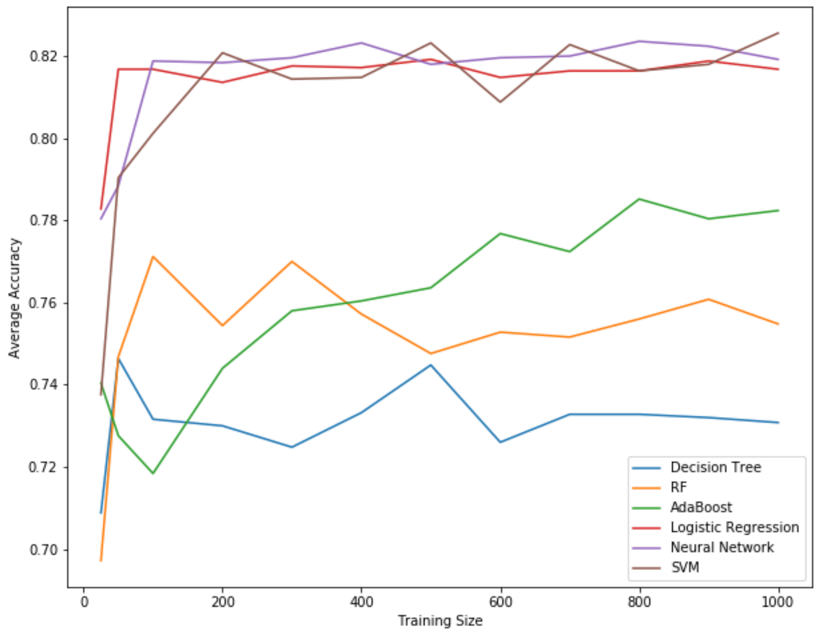
. The code used to generate this plot is:

```
                X, y = create_dataset()
                X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=45)

                mods = [DecisionTreeClassifier(),
                RandomForestClassifier(),
                AdaBoostClassifier(),
                LogisticRegression(),
                MLPClassifier(),
                SVC()]

                titles = ["Decision Tree", "RF", "AdaBoost",
                        "Logistic Regression", "Neural Network", "SVM"]

                fig, ax = plt.subplots(3,2, figsize=(10,10))
                for i, ax in enumerate(ax.flat):
                    mod = mods[i]
                    mod.fit(X_train, y_train)
                    plotter(mod, X, X_test, y_test, titles[i], ax)
                plt.savefig("figures/boundaries.png")
                plt.show()
```

(b) Next, we will study how the performance of each of the clasifiers varies as you increase the size of the training set. Fix your training and test sets from part (a). Then, starting from a random subset (no need to set a seed) of your training set of size 50 (chosen with replacement), train your classification model, and compute the accuracy on the test set. Repeat this process for training set sizes of $[50, 100, 200, 300, \ldots, 1000]$. Repeat the experiment a total of 10 times for each classifier. Then, for each classifier, plot its average accuracy (achieved on the test set) for each training set size. Compare the accuracy across different algorithms in a **single** figure, and in 5 lines or less, discuss your observations: For the models covered in the course so far, use what you know about the bias-variance decomposition to inform your discussion. Which model you prefer for this task?

Please use the following color scheme for your plots: [Decision Tree, Random Forest, AdaBoost, Logistic Regression, Neural Network, SVM], and please include a legend in your plot. *What to submit: a discussion of your observation, a single plot, a print screen of the python code used, a copy of your python code in solutions.py.*

Solution:
The figure generated is



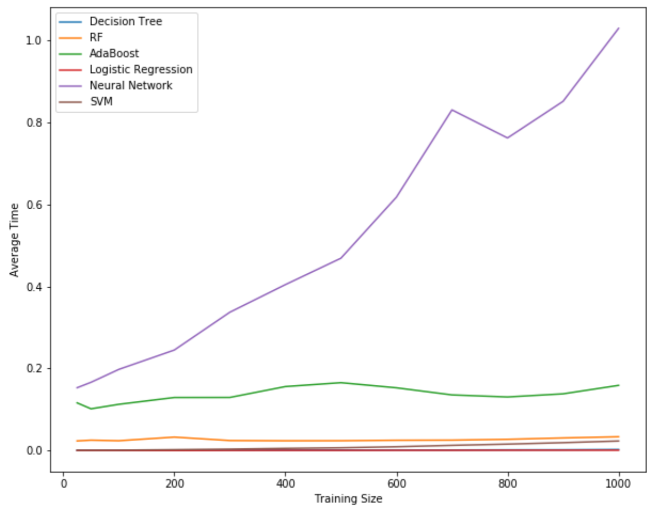
Discussion points can include:

- trees doing poorly and exhibiting high variance, RF seems to be overfitting
- as expected MLPs and SVMs do quite well without much tuning
- logistic regression is preferred here as it the simplest model and achieves consistently good results.

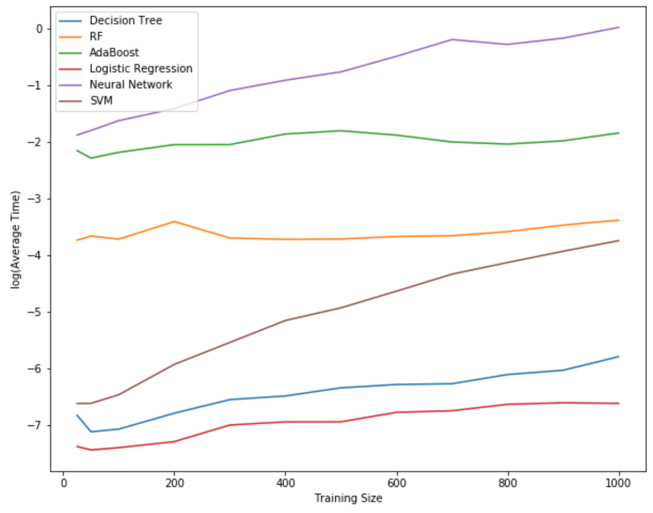
Full marks should use the decision surfaces in (a) to argue about overfitting of RFs and Adaboost relative to the other models. Code is presented at end of part (c).

(c) Using the time module, record the training time for each of the classifiers at each of the training set sizes. Plot the log of the average training time over the 10 trials of each classifier as a function of the training size (use log base e). You may add code for this section to your code in part (b). What do you observe? In 5 lines or less, discuss your observations. Use the same color scheme as in (b). *What to submit: a discussion of your observation, a single plot, a print screen of the python code used, a copy of your python code in solutions.py.*

Solution:
The figure generated is



of if log-scale;



Some basic comments about NNs taking more time as you increase the training size. Some discussion of the default MLP number of hidden layers would be good here but not necessary. Code used for both (b) and (c) is provided here:

```python
def train_mod(mod, X_train, y_train, X_test, y_test, train_size):
    idxs = np.random.choice(X_train.shape[0], size=train_size)
    Xp = X_train[idxs]
    yp = y_train[idxs]
    mod.fit(Xp, yp)
    return np.mean(mod.predict(X_test)==y_test)

mod_mean_accuracies = np.zeros(shape=(6, 12))
mod_mean_time = np.zeros(shape=(6, 12))
train_sizes = [25,50] + [100 + 100*i for i in range(10)]
for mod_idx, mod in enumerate(mods):
    mean_accuracies = np.zeros(shape=(10, 12))
    mean_times = np.zeros(shape=(10, 12))
    for i in range(10):
        for j, tr in enumerate(train_sizes):
            tstart = time.time()
            mean_accuracies[i, j] = train_mod(mod, X_train,
                                              y_train, X_test,
                                              y_test, tr)
            tend = time.time()
            mean_times[i,j] = tend - tstart
    mod_mean_accuracies[mod_idx,:] = mean_accuracies.mean(axis=0)
    mod_mean_time[mod_idx, :] = mean_times.mean(axis=0)

fig_size = plt.rcParams["figure.figsize"]
fig_size[0] = 10
fig_size[1] = 8
plt.rcParams["figure.figsize"] = fig_size

for i in range(6):
    plt.plot(train_sizes, mod_mean_accuracies[i], label=titles[i])
    plt.xlabel("Training Size")
    plt.ylabel("Average Accuracy")
plt.legend()
plt.savefig("figures/Accuracy.png")
plt.show()

for i in range(6):
    plt.plot(train_sizes, mod_mean_time[i], label=titles[i])
    plt.xlabel("Training Size")
    plt.ylabel("Average Time")
plt.legend()
plt.savefig("figures/Timing.png")
plt.show()
```

Question 4

Refer to the following dataset containing a sample S of ten examples. Each example is described using two Boolean attributes A and B. Each is labelled (classified) by the target Boolean function.

A	B	Class
1	0	+
0	1	-
1	1	-
1	0	+
1	1	-
1	1	-
0	0	+
1	1	+
0	0	+
0	0	-

(a) What is the entropy of these examples with respect to the given classification?

Solution:
The classes are split equally (5 in each class), so the entropy is

$$H(S) = -p_+ \log_2 p_+ - p_- \log_2 p_- = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1.$$

(b) What is the Information gain of attribute A on sample S above?

Solution:
If we split on A, we get 6 observations falling into $A = 1$, and they are equally split across the classes. The remaining 4 observations fall in to $A = 0$, and they are also equally split. So they both have entropy of 1. Therefore, the gain of A is

$$G(A) = H(S) - \frac{6}{10} \times 1 - \frac{4}{10} \times 1 = 1 - 1 = 0.$$

In other words, A is useless.

(c) What is the information gain of attribute B on sample S above?

Solution:
In this case, we see 5 observations each in the two regions $B = 1$ and $B = 0$. However, in $B = 1$, we see 1 positive and 4 negatives, so the entropy is $-\frac{1}{5}\log_2\frac{1}{5} - \frac{4}{5}\log_2\frac{4}{5} = 0.7219$. In the region $B = 0$, we see 4 positives and 1 negative, so again the entropy is 0.7219. The gain of B is therefore

$$G(B) = H(S) - \frac{5}{10} \times 0.7219 - \frac{5}{10} \times 0.7219 = 1 - 0.7219 = 0.2781.$$

(d) What would be chosen as the 'best' attribute by a decision tree learner using the ifnromation gain splitting criterion? Why?

Solution:
We choose B as it has higher information gain.

(e) What are ensembles? Discuss one example in which decision trees are used in an ensemble.

Solution:
One example is Bagging trees, for more discussion about this see here.

Question 5

Refer to the following data

x_1	x_2	y
-2	-1	-1
2	-1	1
1	1	1
-1	-1	-1
3	2	1

(a) Apply the Perceptron Learning Algorithm with starting values $w_0 = 5$, $w_1 = 1$ and $w_2 = 1$, and a learning rate $\eta = 0.4$. Be sure to cycle through the training data in the same order that they are presented in the table.

Solution:

Running the Perceptron training algorithm results in the following iterations, where highlighted columns signify that the weight vector is changing on that particular iteration. The correct weights are therefore $w_0 = 3.8, w_1 = 2.6, w_2 = 2.2$. These weights first appear on iteration 9.

Iteration	$w^T x$	$yw^T x$	w
1	2.00	-2.00	$[4.6, 1.8, 1.4]^T$
2	6.80	6.80	$[4.6, 1.8, 1.4]^T$
3	7.80	7.80	$[4.6, 1.8, 1.4]^T$
4	1.40	-1.40	$[4.2, 2.2, 1.8]^T$
5	14.40	14.40	$[4.2, 2.2, 1.8]^T$
6	-2.00	2.00	$[4.2, 2.2, 1.8]^T$
7	6.80	6.80	$[4.2, 2.2, 1.8]^T$
8	8.20	8.20	$[4.2, 2.2, 1.8]^T$
9	0.20	-0.20	$[3.8, 2.6, 2.2]^T$
10	16.00	16.00	$[3.8, 2.6, 2.2]^T$
11	-3.60	3.60	$[3.8, 2.6, 2.2]^T$
12	6.80	6.80	$[3.8, 2.6, 2.2]^T$
13	8.60	8.60	$[3.8, 2.6, 2.2]^T$

(b) Consider a new point, $x_\star = (-5, 3)$. What is the predicted value and predicted class based on your learned perceptron for this point?

Solution:

value: -2.6, class: $y_\star = -1$

(c) Consider adding a new point to the data set, $x_\star = (2, 2)$ and $y_\star = -1$. Will your perceptron converge on the new dataset? How might you remedy this?

(d) Consider the following three logical functions:

1. $A \wedge \neg B$
2. $\neg A \vee B$
3. $(A \vee B) \wedge (\neg A \vee \neg B)$

Which of these functions can a perceptron learn? Explain. What are two ways that you can extend a perceptron to learn all three functions ?

Solution:
A Perceptron can only learn f_1 and f_2. we can extend via multilayer perceptrons, or by using a smart transformation (i.e. kernel perceptron)

Question 6

Refer to the following information. In these two questions you will apply the k-`means` algorithm. You will use a univariate (one-variable) dataset containing the following 12 instances:

$$\{2.01, 3.49, 4.58, 4.91, 4.99, 5.01, 5.32, 5.78, 5.99, 6.21, 7.26, 8.00\}$$

Use the *Manhattan* or *city-block* distance, i.e., the distance between two instances x_i and x_j is the absolute value of the difference $x_i - x_j$. For example, if $x_i = 2$ and $x_j = 3$ then the distance between x_i and x_j is $|2 - 3| = 1$. Use the arithmetic mean to compute the centroids. Apply the k-`means` algorithm to the above dataset of examples. Let $k = 2$. Let the two centroids (means) be initialised to $\{3.33, 6.67\}$. On each iteration of the algorithm record the centroids.

(a) After two iterations of the algorithm you should have recorded two sets of two centroids. What are they?

Solution:
$\{4.00, 6.22\}$ and $\{4.17, 6.43\}$

Now apply the algorithm for one more iteration. Record the new centroids after iteration 3 and answer the following question.

(b) After 3 iterations it is clear that:
(a) due to randomness in the data, the centroids could change on further iterations
(b) due to randomness in the algorithm, the centroids could change on further iterations
(c) k-`means` converges in probability to the true centroids
(d) the algorithm has converged and the clustering will not change on further iterations
(e) the algorithm has not converged and the clustering will change on further iterations

Solution:
After the third iteration, the cluster centroids are $\{4.17, 6.43\}$. This means that there is no change from iteration 2 to iteration 3, so the algorithm has converged, so (d) is true. The reason that (c) is not true is that we cannot be sure that these are the true centroids, so we cannot conclude that the converged centroids are the right centroids with high probability.

Question 7

Note: there will not be any MCQ on the final this year - this question is still useful practice though.
In this question, we will apply a mistake-bounded learner to the following dataset. This dataset has 6 binary features, $x_1, x_2, \ldots x_6$. The class variable y can be either 1, denoting a positive example of the concept to be learned, or 0, denoting a negative example.

Example	x_1	x_2	x_3	x_4	x_5	x_6	**Class**
1)	0	0	0	0	1	1	1
2)	1	0	1	1	0	1	1
3)	0	1	0	1	0	1	0
4)	0	1	1	0	0	1	0
5)	1	1	0	0	0	0	1

Apply the `Winnow2` algorithm to the above dataset of examples **in the order in which they appear**. Use the following values for the `Winnow2` parameters: threshold $t = 2$, $\alpha = 2$. Initialise all weights to have the value 1.

Learned Weights	w_1	w_2	w_3	w_4	w_5	w_6
Weight vector 1	2.000	1.000	1.000	0.000	2.000	1.000
Weight vector 2	3.000	0.000	1.000	1.000	2.000	1.000
Weight vector 3	2.000	2.000	2.000	2.000	2.000	2.000
Weight vector 4	2.000	0.500	0.500	0.500	2.000	0.500
Weight vector 5	2.000	0.250	0.500	0.500	4.000	0.125

(a) After one epoch, i.e., one pass through the dataset, which of the above weight configurations has been learned ?

(a) Weight vector 1
(b) Weight vector 2
(c) Weight vector 3
(d) Weight vector 4
(e) Weight vector 5

Solution:
(d)

(b) On which of the examples did the algorithm ***not*** make a mistake ? (a) Examples 1), 2) and 5)
(b) Example 5)
(c) Example 4)
(d) Examples 4) and 5)
(e) None of the above

Solution:
(e) - the algorithm makes a mistake on each point except for 2).

(c) The algorithm has learned a consistent concept on the training data:

(a) True
(b) False
(c) It is not possible to determine this

Solution:
(a)

(d) Assume the target concept from which this dataset was generated is defined by exactly two features. The worst-case mistake bound for the algorithm on this dataset is approximately:

(a) 1.79
(b) 2.58
(c) 3.58
(d) 4.67
(e) 10.75

Solution:
(c)
comment: The solution is based on the worst case mistake bound $O(r \log n)$ where

$$\begin{aligned} r = & \text{ number of relevant features } = 2 \\ n = & \text{ number of total features } = 6 \end{aligned}$$

So $r \log n = 2 \log 6 = 3.5835$ (log is base e). This concept was introduced on slide 61/78, but also used in Q5 of the learning theory problem set. The idea is that winnow is more robust to irrelevant features, so will scale with the logarithm of the number of irrelevant features. In this case, we are given 6 features $x_1, \ldots, x_6$, but in (d) we are told that the data was actually generated from two features. The proof of the bound is outside the scope of the course but not too complicated, see here for a simplified version.

Question 8

Consider the following dataset:

$$D := \{((1,1,1),0),((1,0,0),0),((1,1,0),1),((0,0,1),1)\},$$

i.e., each data point is composed of a vector of 3 binary features, and a binary response. We will train a decision tree of depth 2 on this dataset using the standard ID3 algorithm covered in lectures and tutorials (please use the natural log (base e) instead of base 2 in all calculations for this question).

(a) Draw the tree computed and write down the training error it achieves. Be sure to show all working for full marks. *What to submit: your working out, either typed or handwritten. Please include a drawing/plot of your tree.*

Solution:
Using ID3, if we split on x_1, there is an information gain of

$$H(\frac{1}{2}) - (\frac{3}{4}H(\frac{2}{3}) + \frac{1}{4}H(0)) \approx 0.22,$$

and there is zero information gain for the other features. So we must pick $x_1 = 0$? as our root. The three examples $(((1,1,1),0),((1,1,0),1),((1,0,0),0)$ will then go down one of the two subtrees, and at this point we must misclassify either of the two points $((1,1,1),0)$ or $((1,1,0),1))$. In any case, we will have at least one point being mislabeled, and so the train error is at least 0.25.

(b) Can you construct a depth 2 decision tree that achieves a lower training error than the one found via ID3? What does this tell you about ID3? Explain. *What to submit: your working out, either typed or handwritten. Please include a drawing/plot of your tree.*

Solution:
We can easily construct a tree that achieves zero training error:

```
x2 = 0:
|     x3 = 0:   0
|     x3 = 1:   1
x2 = 1:
|     x3 = 0:   1
|     x3 = 1:   0
```

So ID3 can be suboptimal (something along the lines of ID3 is a greedy algorithm would be good to mention).

Question 9

Assume that you have access to a function `train_perceptron` that trains a standard perceptron (with learning rate $\eta = 1$, and $w^{(0)} = 0_p$ where 0_p is the vector of zeroes in p dimensions. Here, p denotes the number of features plus 1 for the bias term). Design an algorithm that establishes whether a given dataset is linearly separable or not, and which makes a single call to `train_perceptron`. Describe your algorithm (a few dot points should suffice), and write a Python function implementing the algorithm. When running the perceptron function, terminate it after at most 10000 iterations. Test your function on the following datasets and report whether or not they are linearly separable.

(i) {010, 011, 100, 111}

(ii) {011,100,110,111}

(iii) {0100, 0101, 0110, 1000, 1100, 1101, 1110, 1111}

(iv) {1000000, 1000001, 1000101}.

Present a summary of your results using the following table:

Dataset	Linearly Separable (Yes/No)
(i)	-
(ii)	-
(iii)	-
(iv)	-

The datasets represent the points that are labelled $+1$. All other remaining binary vectors of the same length are labelled -1. For example, for (i), there are $2^3 = 8$ possible binary vectors of length 3, the four strings $\{010, 011, 100, 111\}$ belong to the positive class, the remaining 4 strings $\{000, 001, 101, 110\}$ belong to the negative class. You may find the following StackExchange post helpful in coding up the datasets; it describes a simple approach to generating all length k binary vectors in Python using `itertools.product`.

Note: Using an existing algorithm to determine linear separability here will result in a grade of zero for this part, but you are free to use existing implementations of the perceptron algorithm. *What to submit: a description of your algorithm in words, your filled out table, a screen shot of your code used for this section, a copy of your code in solutions.py*

Solution:

The algorithm is straightforward, run a perceptron on the dataset, and if it converges (set a large enough number of steps), then the data is linearly separable.

Dataset	Linearly Separable (Yes/No)
(i)	No
(ii)	No
(iii)	Yes
(iv)	Yes

```
    import itertools
    import numpy as np

```

```python
    def perceptron(X, y, max_iter=10000):
        np.random.seed(1)
        w = np.zeros(X.shape[1])
        ctr = 0
        for _ in range(max_iter):
            # check which indices we make mistakes on, and pick one randomly to update
            yXw = (y * X) @ w.T
            mistake_idxs = np.where(yXw <= 0)[0]
            if mistake_idxs.size > 0:
                ctr += 1
                i = np.random.choice(mistake_idxs)  # pick idx randomly
                w = w + y[i] * X[i]                 # update w

        return w, ctr, 'separable' if ctr < max_iter else 'not separable'

        # (i)
        X = np.array(list(itertools.product([0, 1], repeat=3)))
        X = np.hstack((np.ones(X.shape[0]).reshape(-1,1), X))
        Y = np.array([-1,-1,1,1,1,-1,-1, 1]).reshape(-1,1)
        w, ctr, iters = perceptron(X, Y, max_iter=100000)
        w, ctr, iters

        # (ii)
        X = np.array(list(itertools.product([0, 1], repeat=3)))
        X = np.hstack((np.ones(X.shape[0]).reshape(-1,1), X))
        Y = np.array([-1,-1,-1,1,1,1,-1,-1]).reshape(-1,1)
        w, ctr, iters = perceptron(X, Y, max_iter=100000)
        w, ctr, iters

        # (iii)
        X = np.array(list(itertools.product([0, 1], repeat=4)))
        X = np.hstack((np.ones(X.shape[0]).reshape(-1,1), X))
        Y = np.array([-1,-1,-1,-1,1,1,1,-1,1,-1,-1,-1,1,1,1,1]).reshape(-1,1)
        w, ctr, iters = perceptron(X, Y, max_iter=100000)
        w, ctr, iters

        # (iv)
        X = np.array(list(itertools.product([0, 1], repeat=6)))
        X = np.hstack((np.ones(X.shape[0]).reshape(-1,1), X))
        Y = -1 * np.ones(X.shape[0])
        Y[0] = 1
        Y[1] = 1
        Y[5] = 1
        Y = Y.reshape(-1,1)
        w, ctr, iters = perceptron(X, Y, max_iter=100000)
        w, ctr, iters
```