COMP9417: A collection of sample exam exercises

July 31, 2021

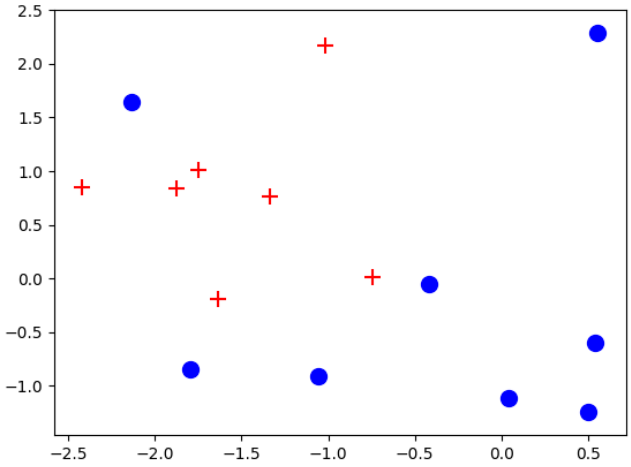
Note to student: Some of these questions are longer/more difficult than the questions that will be on the actual final exam. With that being said, you should aim to work through and understand all questions here as part of your revision. Note that this is not a sample final exam, it is a collection of possible exercises, and the actual final exam will be shorter (more details to be announced later in week 10). Solutions to these problems will be released at the end of Week 10.

Question 1

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

```
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) Comment on your results in (a). What do you notice as you increase the depth of the trees? What do we mean when we say that trees have low bias and high variance?

(c) 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).

(d) 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:

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

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

```

(e) 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.

(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.

(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.

(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

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

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.

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

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

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.

(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:

```
            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()

```

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

(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).

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

(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.

(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.

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.*

(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.*

(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.*

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?

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

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

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

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

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.

(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?

(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 ?

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?

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

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	$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	$\mathbf{x_4}$	$\mathbf{x_5}$	$\mathbf{x_6}$	**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	$\mathbf{w_1}$	$\mathbf{w_2}$	$\mathbf{w_3}$	$\mathbf{w_4}$	$\mathbf{w_5}$	$\mathbf{w_6}$
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

(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 examples

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

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

(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