School of Computing and Information Systems
The University of Melbourne
COMP30027 MACHINE LEARNING (Semester 1, 2019)

Tutorial sample solutions: Week 5

1. For the following dataset:

Label	*apple*	*ibm*	*lemon*	*sun*	CLASS
TRAINING INSTANCES					
A	4	0	1	1	FRUIT
B	5	0	5	2	FRUIT
C	2	5	0	0	COMPUTER
D	1	2	1	7	COMPUTER
TEST INSTANCES					
T_1	2	0	3	1	?
T_2	1	2	1	0	?

   (a) Classify the test instances according to the method of **Nearest Prototype**.

   - We construct a prototype for each class by averaging (taking the mean) of the attribute values, for the instances of that class in the training data:

$$\begin{aligned} P_f &= \langle \frac{4+5}{2}, \frac{0+0}{2}, \frac{1+5}{2}, \frac{1+2}{2} \rangle \\ &= \langle 4.5, 0, 3, 1.5 \rangle \\ P_c &= \langle \frac{2+1}{2}, \frac{5+2}{2}, \frac{0+1}{2}, \frac{0+7}{2} \rangle \\ &= \langle 1.5, 3.5, 0.5, 3.5 \rangle \end{aligned}$$

   - We then classify a test instance according to the closest prototype. We will need to choose a similarity/distance function to do this; I will use Euclidean distance here, but Manhattan is similar:

$$d_E(A, B) = \sqrt{\sum_k (a_k - b_k)^2}$$

   - For the first test instance:

$$\begin{aligned} d_E(T_1, P_f) &= \sqrt{(2-4.5)^2 + (0-0)^2 + (3-3)^1 + (1-1.5)^2} = \sqrt{6.5} \\ d_E(T_1, P_c) &= \sqrt{(2-1.5)^2 + (0-3.5)^2 + (3-0.5)^1 + (1-3.5)^2} = \sqrt{25} \end{aligned}$$

   - The FRUIT prototype is closer, so we will predict that the class of this instance is FRUIT.
   - For the second test instance:

$$\begin{aligned} d_E(T_2, P_f) &= \sqrt{(1-4.5)^2 + (2-0)^2 + (1-3)^1 + (0-1.5)^2} = \sqrt{22.5} \\ d_E(T_2, P_c) &= \sqrt{(1-1.5)^2 + (2-3.5)^2 + (1-0.5)^1 + (0-3.5)^2} = \sqrt{15} \end{aligned}$$

   - This time, the COMPUTER prototype is closer, so we will predict that the class of this instance is COMPUTER.

   (b) Using the **Euclidean distance** measure, classify the test instances using the 1-NN method.

   - This is similar to the previous question, but instead of calculating the distance between a test instance and a prototype, we will calculate the distance between the test instance

and each training instance:

$$\begin{aligned}
d_E(T_1, A) &= \sqrt{(2-4)^2 + (0-0)^2 + (3-1)^2 + (1-1)^2} \\
&= \sqrt{8} \approx 2.828 \\
d_E(T_1, B) &= \sqrt{(2-5)^2 + (0-0)^2 + (3-5)^2 + (1-2)^2} \\
&= \sqrt{14} \approx 3.742 \\
d_E(T_1, C) &= \sqrt{(2-2)^2 + (0-5)^2 + (3-0)^2 + (1-0)^2} \\
&= \sqrt{35} \approx 5.916 \\
d_E(T_1, D) &= \sqrt{(2-1)^2 + (0-2)^2 + (3-1)^2 + (1-7)^2} \\
&= \sqrt{45} \approx 6.708
\end{aligned}$$

- The nearest neighbour is the one with the smallest distance — here, this is instance `A`, which is a FRUIT instance. Therefore, we will classify this instance as FRUIT.
- The second test instance is similar:

$$\begin{aligned}
d_E(T_2, A) &= \sqrt{(1-4)^2 + (2-0)^2 + (1-1)^2 + (0-1)^2} \\
&= \sqrt{14} \approx 3.742 \\
d_E(T_2, B) &= \sqrt{(1-5)^2 + (2-0)^2 + (1-5)^2 + (0-2)^2} \\
&= \sqrt{40} \approx 6.325 \\
d_E(T_2, C) &= \sqrt{(1-2)^2 + (2-5)^2 + (1-0)^2 + (0-0)^2} \\
&= \sqrt{11} \approx 3.317 \\
d_E(T_2, D) &= \sqrt{(1-1)^2 + (2-2)^2 + (1-1)^2 + (0-7)^2} \\
&= \sqrt{49} = 7
\end{aligned}$$

- Here, the nearest neighbour is instance `C`, which is a COMPUTER instance. Therefore, we will classify this instance as COMPUTER.

(c) Using the **Manhattan distance** measure, classify the test instances using the 3-NN method, for the three weightings we discussed in the lectures: majority class, inverse distance, inverse linear distance.

- The first thing to do is to calculate the Manhattan distances, which is like the Euclidean distance, but without the squares/square root:

$$d_M(A, B) = \sum_k |a_k - b_k|$$

$$\begin{aligned}
d_E(T_1, A) &= |2-4| + |0-0| + |3-1| + |1-1| = 4 \\
d_E(T_1, B) &= |2-5| + |0-0| + |3-5| + |1-2| = 6 \\
d_E(T_1, C) &= |2-2| + |0-5| + |3-0| + |1-0| = 9 \\
d_E(T_1, D) &= |2-1| + |0-2| + |3-1| + |1-7| = 11 \\
d_E(T_2, A) &= |1-4| + |2-0| + |1-1| + |0-1| = 6 \\
d_E(T_2, B) &= |1-5| + |2-0| + |1-5| + |0-2| = 12 \\
d_E(T_2, C) &= |1-2| + |2-5| + |1-0| + |0-0| = 5 \\
d_E(T_2, D) &= |1-1| + |2-2| + |1-1| + |0-7| = 7
\end{aligned}$$

- The nearest neighbours for the first test instance are `A`, `B`, and `C`. For the second test instance[1], they are `C`, `A`, and `D`.

[1] If you compare to the Euclidean distance to the Manhattan distance, the order of the neighbours for the second test instance are slightly different.

- The **majority class** weighting method effectively assigns a weight of 1 to every instance in the set of nearest neighbours:
  - For the first test instance, there are 2 FRUIT instances and 1 COMPUTER instance. There are more FRUIT than COMPUTER, so we predict FRUIT.
  - For the second test instance, there are 2 COMPUTER instances and 1 FRUIT instance. There are more COMPUTER than FRUIT, so we predict COMPUTER.
- To use the **inverse distance** weighting, we first need to choose a value for ϵ, let's say 1:
- For the first test instance:
  - The first neighbour (a FRUIT) gets a weight of $\frac{1}{d+\epsilon} = \frac{1}{4+1} = 0.2$
  - The second neighbour (a FRUIT) gets a weight of $\frac{1}{d+\epsilon} = \frac{1}{6+1} \approx 0.14$
  - The third neighbour (a COMPUTER) gets a weight of $\frac{1}{d+\epsilon} = \frac{1}{9+1} = 0.1$
- Overall, FRUIT instances have a score of $0.2 + 0.14 = 0.34$, and COMPUTER instances have a score of 0.1, so we would predict FRUIT for this instance.
- For the second test instance:
  - The first neighbour (a COMPUTER) gets a weight of $\frac{1}{d+\epsilon} = \frac{1}{5+1} \approx 0.17$
  - The second neighbour (a FRUIT) gets a weight of $\frac{1}{d+\epsilon} = \frac{1}{6+1} \approx 0.14$
  - The third neighbour (a COMPUTER) gets a weight of $\frac{1}{d+\epsilon} = \frac{1}{7+1} = 0.12$
- Overall, FRUIT instances have a score of 0.14, and COMPUTER instances have a score of $0.17 + 0.12 = 0.29$, so we would predict COMPUTER for this instance[2].
- How about **inverse linear distance**? We are going to weight instances by re-scaling the distances according to the following formula, where d_j is the distance of the j^{th} nearest neighbour[3]:

$$w_j = \frac{d_3 - d_j}{d_3 - d_1}$$

- For the first test instance, the 1st nearest neighbour is at a distance of 4, and the 3rd is at a distance of 9:
  - The first neighbour (a FRUIT) gets a weight of $\frac{d_3-d_1}{d_3-d_1} = \frac{9-4}{9-4} = 1$
  - The second neighbour (a FRUIT) gets a weight of $\frac{d_3-d_2}{d_3-d_1} = \frac{9-6}{9-4} = 0.6$
  - The third neighbour (a COMPUTER) gets a weight of $\frac{d_3-d_3}{d_3-d_1} = \frac{9-9}{9-4} = 0$
- Overall, FRUIT instances have a score of $1 + 0.6 = 1.6$, and COMPUTER instances have a score of 0, so we would predict FRUIT for this instance.
- For the second test instance, the 1st nearest neighbour is at a distance of 5, and the 3rd is at a distance of 7:
  - The first neighbour (a COMPUTER) gets a weight of $\frac{d_3-d_1}{d_3-d_1} = \frac{7-5}{7-5} = 1$
  - The second neighbour (a FRUIT) gets a weight of $\frac{d_3-d_2}{d_3-d_1} = \frac{7-6}{7-5} = 0.5$
  - The third neighbour (a COMPUTER) gets a weight of $\frac{d_3-d_3}{d_3-d_1} = \frac{7-7}{7-5} = 0$
- Overall, FRUIT instances have a score of 0.5, and COMPUTER instances have a score of $1 + 0 = 1$, so we would predict COMPUTER for this instance.

(d) Can we do weighted k-NN using **cosine similarity**?

- Of course! If anything, this is easier than with a distance, because we can assign a weighting for each instance using the cosine similarity directly. An overall weighting for a class can be obtained by summing the cosine scores for the instances of the corresponding class, from among the set of nearest neighbours.
- Let's summarise all of these predictions in a table (overleaf). We can see that there is some divergence for these methods, depending on whether B or D is the 3rd neighbour for T_2:

[2]Note that the Euclidean distance would give a different result here!

[3]Compared to the lecture version, we have substituted $k = 3$ here, because we are using the 3-Nearest Neighbour method.

Inst	Measure	k	Weight	Prediction
T_1	d_E	1	-	FRUIT
		3	Maj	FRUIT
		3	ID	FRUIT
		3	ILD	FRUIT
	d_M	1	-	FRUIT
		3	Maj	FRUIT
		3	ID	FRUIT
		3	ILD	FRUIT
	cos	1	-	FRUIT
		3	Maj	FRUIT
		3	Sum	FRUIT
T_2	d_E	1	-	COMPUTER
		3	Maj	FRUIT
		3	ID	FRUIT
		3	ILD	COMPUTER
	d_M	1	-	COMPUTER
		3	Maj	COMPUTER
		3	ID	COMPUTER
		3	ILD	COMPUTER
	cos	1	-	COMPUTER
		3	Maj	FRUIT
		3	Sum	FRUIT

2. Revise SVMs, particularly the notion of "linear separability".

   (a) If a dataset isn't linearly separable, an SVM learner has two major options. What are they, and why might we prefer one to the other?

   - **Soft margins**: we permit a few points to be on the "wrong" side of the line, at a penalty, if this allows us to find a better (wider) margin.
   - **Kernel methods**: we (effectively) transform the data into a higher-dimensional space. Conceptually, this is by creating new dimensions (and consequently, new attributes), but the "kernel trick" allows us to do this without making the transformation explicit (and so, saving us some computation time).
   - If we suspect that the data is essentially linearly separable, except that a few points will be mis-classified (for example, if there are **outliers** in the data), then using a soft-margin SVM is the better choice. The kernel trick is too powerful to account for just a few instances (and, in the worst case, will generate a spurious transformation of the data to deal with them).
   - If, on the other hand, we suspect that the data isn't really linearly separable — because the instance topology isn't linear, but polynomial or circular or something — then the soft margin SVM will produce a spurious margin, because there really isn't a way to draw a straight line through the data at all. (This can be confirmed empirically, by cross-validating, at the cost of some time.)
   - One major concern, separate to these notions is that the most popular method for building an SVM (SMO) converges **much** more quickly with a linear kernel than with a non-linear kernel. (This can be seen empirically.) Consequently, unless you have **a lot** of time to wait for the training cycles, non–linear kernels are generally only used as a last resort. (And many problems don't really require them anyway!)

   (b) Contrary to many geometric methods, SVMs work better (albeit slower) with large attribute sets. Why might this be true?

   - The clearest way of seeing this is to imagine that the dataset has a number of useful attributes (as in, they are strongly or moderately predictive of the class), and a number of not-so-useful attributes (as in, they are marginally predictive, or not predictive at all).

This is actually typical of many problem formulations, and is one of the main drivers of **feature selection**, which we will see later in semester.

- Most geometric methods calculate some kind of **similarity** or **distance** metric, which assumes that every attribute is equally important. For example, using the Manhattan Distance, the difference between the values of the useful attributes can be small, but if the difference between the not-so-useful attributes is large, then the two instances will be not-so-similar (they will be far apart in space)[4].
- A typical SVM gets around this problem by (effectively) finding different "weights" for each attribute[5] — these form the normal vector w; the weights of the not-so-useful attributes will (hopefully) be negligible, compared to the weights of the useful attributes, so that the basis for the prediction (hopefully) becomes more meaningful.
- There are other issues too, like how the SVM is building a linear combination (or non-linear combination, if we are using a non-linear kernel) of the attributes, rather than treating them all as completely distinct.

3. We have now seen a decent selection of (supervised) learners:

   - Naive Bayes
   - 0-R
   - 1-R
   - Decision Trees
   - k-Nearest Neighbour
   - Nearest Prototype
   - Support Vector Machines

   (a) For each, identify the model built during training.
   - For NB, this is a set of prior probabilities $P(c_j)$ and a set of posterior probabilities $P(a_k|c_j)$
   - For 0-R, this is a class distribution, or simply the label of the most frequent class
   - For 1-R, this is an attribute, and the majority class of the instances (a label) for each value that this attribute can take
   - For a Decision Tree, this is the tree itself: every non-terminal node is labelled with an attribute, and each branch with an attribute value; every leaf is labelled with a class
   - For k-NN, this is just the dataset itself!
   - For NP, this is a prototype (vector) for each class
   - For an SVM, this is the maximum-margin hyperplane, defined by w and b, or equivalently by a set of Lagrance multipliers α_k, and the corresponding training instances for which $\alpha_k > 0$.

   (b) Rank the learners (approximately) by how fast they can classify a large set of test instances. (Note that this is largely independent of how fast they can build a model, and how well they work in general!)
   - Given N training instances, C classes, and D attributes, the amount of time required to make a prediction for each test instance would be as follows:
   - The fastest few are clear: 0-R ($\mathcal{O}(0)$), 1-R ($\mathcal{O}(1)$), and Decision Trees ($\mathcal{O}(D)$).
   - The next two are essentially equally fast: NP ($\mathcal{O}(CD)$) and NB ($\mathcal{O}(CD + C)$).
   - An SVM isn't built to deal with multiple classes directly. If there are exactly two classes, it is just as fast as the previous two, which in turn are essentially indistinguishable from a Decision Tree. If there are many classes, and we are using a one-vs-all strategy for building our SVM(s), it is approximately the same as NB ($\mathcal{O}(CD + C)$). If we are using one-vs-one, it is somewhat slower ($\mathcal{O}(C^2D + C^2)$).
   - k-NN is much, much slower, because it must make a comparison for each instance ($\mathcal{O}(ND + k)$).

[4] In fact, as the attribute set becomes large, there is a statistical bias to the distance measures producing values that make instances indistinguishable! In ML, this is known as the **curse of dimensionality**.

[5] There are improvements to k-NN which attempt to follow a similar logic, creating a vector space that isn't orthonormal.