# Solution to the exam
# DIT866/DAT340: Applied Machine Learning, March 18–20, 2020

## Question 1 of 12: Predicting fuel consumption in ships (8 points)

A shipping company would like to build a machine learning model that predicts the fuel consumption of a ship, given a number of continuous and discrete measurements such as the ship's speed, the engine's RPM (revolutions per minute), wind draft measurements, etc.

A ship travels from Southampton in the UK to Port Elizabeth in South Africa without stopping in any seaports along the way. We collect sensor readings every 5 minutes during this voyage, and also record how much fuel has been consumed during each 5-minute interval. This is a 16-day trip, so we end up with a few thousand examples in our dataset.

**(a, 6p)** Explain how you would implement a machine learning model for the fuel consumption prediction task. You don't need to show Python code, but please give a description of the system and explain all steps you would carry out when developing it.

**(b, 2p)** Discuss the limitations of the approach that we have used here.

**Solution.**

**(a)** This task is inspired by some research at the Marine Technology department at Chalmers.

Here, the idea is that you describe the whole development process required when applying a machine learning approach. We will involve scikit-learn in the explanations here, but this is not compulsory as long as the explanation is clear.

We first convert the feature values into a matrix. We have a mix of discrete and continuous features here, so we should probably use some sort of vectorizer that will encode the discrete features as one-hot vectors. We can use a `DictVectorizer` for this purpose, the `ColumnTransformer`, or alternatively use the `pandas` library.

This is clearly a regression problem, so we need to pick a useful regression model. We should probably try several models, including linear (e.g. `Ridge`), tree ensembles (`GradientBoostingRegressor`, `RandomForestRegressor`), and neural (`MLPRegressor`).

We should try out different ways to preprocess features. For instance, with numerical features it is often useful to standardize features (`StandardScaler`). It can also be useful to apply a feature selection method (`SelectKBest`). If you have some knowledge of the domain, it will probably be useful at this stage.

In terms of experimental setup, we should reserve a test set. We select the best model (learning method, preprocessing, hyperparameter tuning) either by also using a development set, or via cross-validation. In scikit-learn, we have `GridSeachCV` and `RandomSearchCV` to help us tune parameters, for instance. Finally, we evaluate on the test set. The evaluation metric will be a regression metric such as mean squared error (scikit-learn: `mean_squared_error`) or $R^2$ (`r2_score`). If there is an established experimental protocol in this field, we should probably use that as well.

It can be useful to compare our result to a baseline, such as a `DummyRegressor`, to make sure that we are actually doing something useful. This is particularly important if there's an existing system that we are comparing to, although that is not mentioned in this case.

**(b)** In machine learning, we assume that the circumstances at the deployment stage will be similar to what they were during the data collection. This assumption is probably false here, since the data has been collected with a single ship during a single voyage. It seems likely that the regression model developed here will generalize quite poorly, because next time we may have a different ship, different cargo, different type of fuel, etc.

## Question 2 of 12: Training a neural network (3 points)

We use Keras to train a neural network for a binary classification task. The diagnostic output printed by Keras during training (100 epochs) is shown below.

```
Epoch   5: loss: 0.3206 - acc: 0.8499 - val_loss: 0.3289 - val_acc: 0.8460
Epoch  10: loss: 0.2962 - acc: 0.8614 - val_loss: 0.3219 - val_acc: 0.8477
Epoch  15: loss: 0.2806 - acc: 0.8695 - val_loss: 0.3216 - val_acc: 0.8502
Epoch  20: loss: 0.2681 - acc: 0.8742 - val_loss: 0.3284 - val_acc: 0.8502
Epoch  25: loss: 0.2557 - acc: 0.8812 - val_loss: 0.3311 - val_acc: 0.8498
Epoch  30: loss: 0.2448 - acc: 0.8856 - val_loss: 0.3415 - val_acc: 0.8484
Epoch  35: loss: 0.2345 - acc: 0.8907 - val_loss: 0.3516 - val_acc: 0.8465
Epoch  40: loss: 0.2250 - acc: 0.8942 - val_loss: 0.3608 - val_acc: 0.8456
Epoch  45: loss: 0.2168 - acc: 0.8985 - val_loss: 0.3766 - val_acc: 0.8428
Epoch  50: loss: 0.2098 - acc: 0.9015 - val_loss: 0.3846 - val_acc: 0.8394
Epoch  55: loss: 0.2023 - acc: 0.9052 - val_loss: 0.3946 - val_acc: 0.8380
Epoch  60: loss: 0.1950 - acc: 0.9087 - val_loss: 0.4110 - val_acc: 0.8383
Epoch  65: loss: 0.1893 - acc: 0.9113 - val_loss: 0.4182 - val_acc: 0.8386
Epoch  70: loss: 0.1827 - acc: 0.9147 - val_loss: 0.4317 - val_acc: 0.8360
Epoch  75: loss: 0.1778 - acc: 0.9175 - val_loss: 0.4427 - val_acc: 0.8338
Epoch  80: loss: 0.1714 - acc: 0.9200 - val_loss: 0.4528 - val_acc: 0.8322
Epoch  85: loss: 0.1671 - acc: 0.9222 - val_loss: 0.4641 - val_acc: 0.8322
Epoch  90: loss: 0.1626 - acc: 0.9242 - val_loss: 0.4729 - val_acc: 0.8372
Epoch  95: loss: 0.1580 - acc: 0.9263 - val_loss: 0.4855 - val_acc: 0.8317
Epoch 100: loss: 0.1538 - acc: 0.9286 - val_loss: 0.4934 - val_acc: 0.8327
```

After training, we evaluate the classifier on a separate test set and get an accuracy of 0.82. How do you think we can improve the test set accuracy somewhat?

**Solution.**

If you look at the validation-set accuracy, and the corresponding loss, it is clear that we have quite a bit of overfitting here: after training for 100 epochs, the validation-set accuracy has dropped by almost 2 absolute points from its peak. If we use early stopping (or manually terminate the training after 20 epochs), the accuracy on the validation set is significantly higher, so we can probably expect the same to be true for the test set.

Alternatively, we can consider other types of regularizers commonly used in neural networks, including dropout and $L_2$ regularization ("weight decay").

## Question 3 of 12: Blood pressure prediction (7 points)

Let's assume, unrealistically, that we have built a machine learning system that predicts the systolic blood pressure level of an individual by applying a convolutional neural network to an image of the person's face.

**(a, 2p)** We collect a small test set to see how the system's predictions relate to the true blood pressure values. The table below shows the result.

| True value | Predicted value |
|---|---|
| 131 | 129 |
| 142 | 133 |
| 105 | 101 |
| 147 | 142 |
| 120 | 100 |
| 90 | 86 |
| 114 | 100 |
| 145 | 152 |
| 138 | 137 |
| 101 | 93 |

How should we evaluate this system? Select an evaluation metric and compute it for this example.

**(b, 2p)** Define a suitable trivial baseline and evaluate it in the same way. (We don't see the training data here, but you may assume that the blood pressure measurements in the training data for this baseline are "similar" to the test data.)

**(c, 3p)** Let's say that we consider levels of systolic blood pressure greater than the threshold of 140 to be abnormally high (*hypertension*). If we use this system for the purpose of finding the people with abnormally high blood pressure, how should we evaluate it? Compute the relevant scores.

**Solution.**

**(a)** This is a regression problem (we're predicting numerical values) so we should use some sort of regression evaluation metric, typically the *mean squared error*, *mean absolute error*, or $R^2$. We skip the calculations here; the evaluation scores are: MSE = 85.2, MAE = 7.4, $R^2$ = 0.77. Recall that we'd like our MSE and MAE scores to be low and the $R^2$ to be high.

**(b)** We may think of different trivial baselines, but the most natural one is probably to use a constant regressor that always outputs the mean value of this dataset. This is equivalent to `DummyRegressor` in scikit-learn. If we assume that the training set is "similar" to the test set, the mean value is 123.3. If all predictions are equal to this mean value, the evaluation scores are: MSE = 369.6, MAE = 17.3, $R^2$ = 0. Note that the $R^2$ score will be close to zero for this type of baseline.

**(c)** We now view this as a classification task of the "spotting" or "needle in haystack" kind: we're trying to find the people with hypertension among the healthy population. For this type of scenario, especially if the unhealthy people are rare, classification accuracy makes little sense since a dummy baseline would have a high accuracy (0.7 in this case). Sensible choices here are precision/recall, ROC curve, Average precision, etc. We keep things simple here and compute just the precision and recall, which are

$$P = \frac{2}{2} \qquad R = \frac{2}{3}$$

If you still insist on using the accuracy, the result will be 0.9.

## Question 4 of 12: Machine learning in insurance (3 points)

An insurance company develops two machine learning systems: a binary classifier that determines whether or not to offer an insurance plan to a potential customer, and a regression system that suggests a premium. The features used in these prediction systems will obviously depend on what type of insurance policy we are considering: for instance, for a car insurance we may consider the age and the traffic history of the driver, etc.

Explain why it is probably more useful for the company to use fairly small decision trees rather than random forests or neural networks when we develop these prediction systems.

**Solution.**

It is of course arguable whether it is a good idea at all to use an automatic system in this scenario, but if the company chooses to do so, it is probably useful to select machine learning models that give more interpretable predictions.

With a simple decision tree, we can directly read the classifier's "decision process" off the tree: *we won't offer you a car insurance plan, since you had an accident within the last two years* etc. This type of rationale is more difficult to produce with NNs or RFs. This is of course under the assumption that we can reach a quality of predictions that is "good enough" compared to the more complex models.

Note also that recent EU regulations require companies that use automatic decision systems that affect humans to produce a rationale for their decision if asked to do so.

## Question 5 of 12: Different types of regression models (7 points)

For a regression task, let us think of how different types of models may be more or less suitable, depending on the "shape" of the dataset.

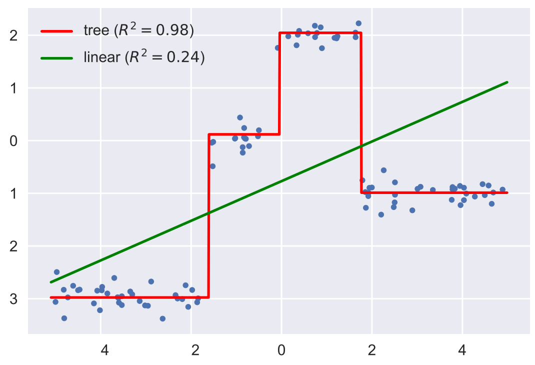
**(a, 2p)** What kind of dataset is ideally suited for a decision tree regression model but poorly suited for a linear model?

**(b, 2p)** Conversely, what kind of dataset is ideally suited for a linear regression model but poorly suited for a decision tree?

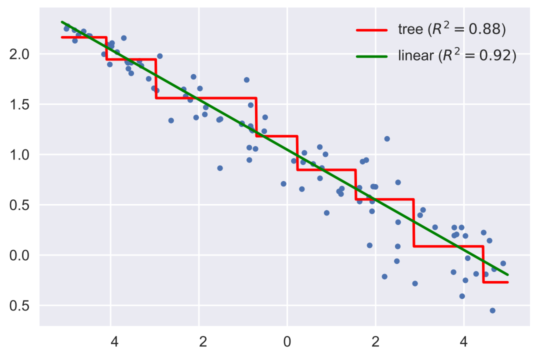
**(c, 3p)** How do you think neural network regression models behave with respect to the datasets you discussed in (a) and (b)?

**Solution.**

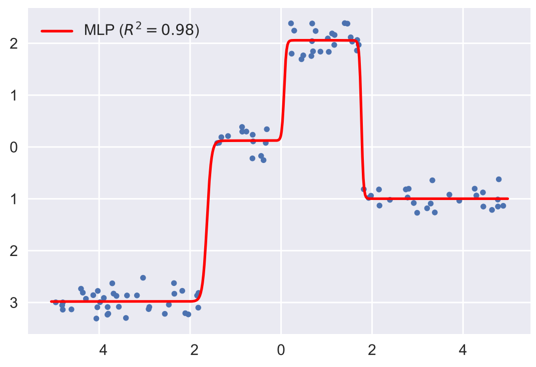
**(a)** Decision tree regression models assume that the data is piecewise constant, so the ideal dataset for decision trees will consist of regions that are more or less flat. If there are abrupt "jumps" between the flat regions, such datasets will be difficult to handle for linear models, although we can mitigate this problem somewhat by adding higher-order terms ($x^2$, $x^3$ etc) to the feature matrix: see `PolynomialFeatures`. The following synthetic dataset gives an illustration:

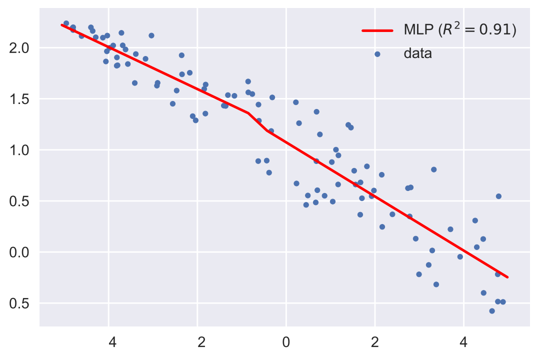


**(b)** Linear regression models assume that the data is more or less line-shaped. Decision tree regression models will need to approximate this line with a "staircase" shape. Of course, if you use a very deep tree, you can have very small steps in this "staircase," but as we have seen previously we are likely to see problems with overfitting if we let the trees grow too deep. Here is an illustration:



**(c)** Neural networks should in principle be able to handle the two types of datasets we discussed in (a) and (b), but we need to make sure that we tune carefully. The "jumps between flat regions" shape that we saw in (a) can be modeled if we use sigmoid or tanh activations in the hidden layer. And (b) is kind of trivial, if we see a linear model as a special case of a neural network. (More generally, if we use ReLU activation we get a piecewise linear model.)





## Question 6 of 12: Machine learning in a car safety system (8 points)

In a car manufacturing company, we have developed a machine learning system that determines whether the car is skidding or not. This classifier is based on measurements read from 20 different sensors and has been trained on a large volume of historical data.

**(a, 3p)** After developing and training this classifier, the software in some of the sensors has been found to be faulty. The subcontractors that deliver the sensors release updates that correct these software errors. Please explain what consequences you can expect in your machine learning system and what you can do about them.

**(b, 2p)** In the car's safety system, our classifier is not used on its own, but is combined with two other classifiers (one based on a rule system using the same sensor values, and another on specialized hardware). The final decision is carried out by computing a weighted formula that uses the outputs from all the three classifiers. What might have motivated us to use this combination of three systems instead of just using one?

**(c, 3p)** Our machine learning classifier was implemented as a neural network and we have found that we can improve the accuracy slightly by switching to a tree-based gradient boosting classifier. How might this affect the combined system described in (b)?

**Solution.**

These questions, particularly (a) and (c), try to illustrate some challenges of using ML in large systems involving many interconnected components.

**(a)** The exact answer probably depends on the nature of the error that we had in our previous data, but it seems fairly likely that the *distribution* of these sensor readings will change when update the software in the sensors. If the measurements were skewed when we trained the system, but not skewed when we use the system, we can expect it to behave worse after the update even if the measurements are more accurate. (This is under the assumption that the features involving the faulty sensors are actually useful for the model.)

In the ideal case, we would like to re-generate the training dataset using the correct sensor measurements instead of the faulty ones and then retrain the model from scratch. In practice, if the training data was recorded in some way that cannot be reproduced easily, then we may try to train a model that "adjusts" the old sensor values so that they resemble values from the updated sensors. The latter option is likely to be less reliable.

Another option would be to retrain the model, removing the features involving the faulty sensors. If these features were not particularly useful, maybe this is the easiest option.

**(b)** In this particular case, it may be that the three different systems have complementary strengths which makes a combined system more robust. More generally, this is an example of an *ensemble* system, which often perform more robustly than single systems.

**(c)** This question is closely related to (a): while previously we had a problem that the *input* distribution might change, now it is the *output* distribution that changes. The "weighted formula" used in the ensemble has obviously been designed for the neural network, not the GB model. So when we switch to the GB classifier, it might be the case that the ensemble starts to behave more poorly even if the accuracy of the single model has improved. Again, it is probably best if we can retrain the model (that is, adjust the ensemble's weights).

## Question 7 of 12: Transfer learning (5 points)

Describe the use of *transfer learning* in image classification problems and related tasks. How do these approaches work technically and what are the advantages? When do we think that transfer learning may or may not be applicable?

**Solution.**

Transfer learning in general means that we try to reuse what we learned in one learning task in some new task. In image classification it is common, as we saw in Programming Assignment 3, to take a model trained on some fairly large-scale image classification task (most commonly the ImageNet dataset) and "refit" it in some way for a new classification task.

The selling point for this "refitting" of an existing classifier is that we hope that the previous classifier has learned some abstractions that are useful in the new classification task as well, which would then reduce the need for training data compared to training the new classifier from scratch. This is particularly useful if the amount of data available for the target task is small.

For image classification tasks, we normally use a convolutional neural network (CNN) as the classification model, and in this case the transfer learning approach will most commonly be based on taking parts of the trained CNN and reusing them in the new classifier. In the simplest case, we just replace the topmost layer of the CNN and retrain this layer on the new task: this was exactly what we did in Programming Assignment 3. More sophisticated approaches modify ("fine-tune") the pre-trained architecture in some way, for instance by using different learning rates for higher and lower layers.

Generally, it is an empirical (trial and error) question whether a transfer learning approach will work in a given target task. Intuitively, we expect transfer learning to be more applicable if the target task resembles the original task somewhat, but in practice transfer learning from e.g. ImageNet can often work surprisingly well in tasks where the images do not resemble the ImageNet dataset.

## Question 8 of 12: Variations of random forests (5 points)

In decision tree models and in tree ensembles such as random forests, when the data includes *numerical* features, each tree node typically involves a comparison to a threshold value, and the left or right branch is selected depending on whether the feature's value is greater than or less than this threshold. As we have seen in lectures and assignments, in the standard decision tree learning algorithm, we find the *best* threshold for each feature by computing a homogeneity criterion with splits defined by different thresholds.

We would like to try a variant of the random forest learning algorithm, with a small twist: when we consider a feature that takes numerical values, we will use a *randomly selected* threshold instead of the *best* threshold. The threshold is selected uniformly randomly between the minimal and maximal value of the feature in the training set.

**(a, 1p)** Why do we end up with trees in the ensemble that are different from each other? Describe all the ways in which randomness affects the training algorithm in this case.

**(b, 2p)** How do you think this change will affect the learning algorithm's behavior compared to the standard random forest algorithm?

**(c, 2p)** Invent a novel type of random forest where you introduce randomness into the training process in some new way.

**Solution.**

This training approach is called *Extremely Randomized Trees* or ExtraTrees. In scikit-learn, we have implementations called `ExtraTreesClassifier` and `ExtraTreesRegressor`.

**(a)** In this training algorithm, we have *three* sources of randomness:

- *training set*: in RF training, we "bootstrap" a training set by sampling uniformly with replacement from the original training set;
- *feature selection*: in RF training, when building a decision tree node, we consider a randomly selected subset from the full set of features;
- *thresholds*: this is in the current modified RF training, and as described in the question, we now define a splitting threshold randomly instead of selecting the optimal threshold.

**(b)** As usual the true answer is "it depends" on the dataset and other factors. As we have seen in one of the assignments, a single decision tree from a random forest performs quite poorly because of the noise introduced during training, so it seems likely that each individual tree will behave even more poorly in this case. This probably means that we will need a larger number of trees to get a good performance. On the other hand, with all the randomness we use during training, the new algorithm is probably less prone to overfitting.

One clear and unambiguous difference is in the computational complexity: if we use the same number of trees as in a standard random forest, the new algorithm will be faster because we don't have to search for the best threshold, which is an operation that has linear time complexity with respect to the number of training instances.

**(c)** Here, you can let your creativity flow and there will not be a finite set of correct solutions. (We will accept most solutions as long as they are well motivated and seem more or less reasonable.) Here are some examples of ways we can introduce more randomness in the training process:

- *noise / data augmentation*: as we saw in the CNN assignment, it can sometimes be useful to modify the training instances during training, and we can imagine that we add some noise to the data after the bootstrapping step or when building the nodes;
- *tree depth*: we may decide to terminate the recursion in decision tree training depending on a random decision variable;
- *ranking criterion*: we can add some random noise to the output of the homogeneity criterion, or we may select randomly between different homogeneity criteria.

## Question 9 of 12: Neural networks and related models (6 points)

Please answer the following two questions about neural networks and other types of models that are related to them.

**(a, 3p)** We train a binary classifier using scikit-learn as follows:

```
from sklearn.ensemble import BaggingClassifier
from sklearn.linear_model import LogisticRegression

X, Y = ... some training set ...

clf = BaggingClassifier(LogisticRegression(), n_estimators=10)
clf.fit(X, Y)
```

Show that there is a neural network that gives exactly the same output as this classifier.

**Hint.** Predictions in a `BaggingClassifier` are computed by averaging the probabilities computed by the component classifiers if they are probabilistic (that is, if they have a method called `predict_proba`). Otherwise, voting will be used.

**(b, 3p)** A *linear model tree* is a decision tree that keeps a full linear (classification or regression) model at every leaf node, in contrast to standard decision trees where the leaf nodes represent constant values.

For a neural network regression model with a one-variable input and a single hidden layer with rectified linear units, explain how to construct a linear model tree that gives the same output as the neural network for all inputs.

**Solution.**

**(a)** Scitkit-learn's `BaggingClassifier` will train an ensemble of classifiers, in this case 10 logistic regression classifiers. These classifiers are trained on different bootstrapped samples from the original training set. As described in the hint, the output of the ensemble is then computed by averaging the probability output from each of the LR classifiers.

A "sigmoid unit" in a neural network's hidden layer is technically identical to a LR classifier: both apply the sigmoid function to the result of a linear operation $\boldsymbol{w} \cdot \boldsymbol{x}$.

So we can view this ensemble as a neural network with a hidden layer consisting of 10 sigmoid units. The output consists of a linear layer with all weights set to 1/10.

**(b)** A neural network regression model with a ReLU hidden layer behaves as a piecewise linear regression model: there are some thresholds where the output curve's direction changes abruptly. Each threshold corresponds to a point where a ReLU unit is activated or deactivated. Between these thresholds, the curve is completely straight (because the ReLU is zero or linear). Informally, what we want to do is to build a decision tree that uses these thresholds, and each node will contain a linear model that describes a line segment.

So a systematic procedure to build this linear model tree could be something like the following, written briefly:

- find thresholds $t_1 < \ldots < t_n$ for all the ReLUs
- build linear regressors that model each line segment:
  - $m_0$ for the segment $-\infty$ to $t_1$
  - $m_1$ for the segment $t_1$ to $t_2$
  - $\ldots$
  - $m_{n-1}$ for the segment $t_{n-1}$ to $t_n$
  - $m_n$ for the segment $t_n$ to $\infty$
- build a decision tree that determines for each $x$ which line segment to use

## Question 10 of 12: Training a linear classifier (7 points)

We would like to train a binary linear classifier that tries to minimize the following loss function:

$$\text{Loss}(\boldsymbol{w}, \boldsymbol{x}, y) = \max(0, -y \cdot \boldsymbol{w} \cdot \boldsymbol{x})$$

As usual, in this formula $\boldsymbol{x}$ is a feature vector representing the instance for which we are making a prediction and $\boldsymbol{w}$ is the model's weight vector. $y$ is a number representing the output class, coded as $+1$ or $-1$.

The gradient (or more precisely, a subgradient) of this loss function with respect to $\boldsymbol{w}$ is

$$\nabla_{\boldsymbol{w}} \text{Loss} = \begin{cases} -y \cdot \boldsymbol{x} & \text{if } y \cdot \boldsymbol{w} \cdot \boldsymbol{x} \leq 0 \\ (0, \ldots, 0) & \text{otherwise} \end{cases}$$

**(a, 4p)** Write the pseudocode (or Python approximation) for a stochastic gradient descent algorithm (with a minibatch size of 1) to train a classifier by minimizing this loss function on a training set. You can assume that the learning rate is constant and that we don't use a regularizer.

**(b, 1p)** Assuming your solution in (a) is correct, you have re-created a well-known machine learning algorithm. What is it called?

**(c, 2p)** Add an $L_2$ regularizer to the model. How does your pseudocode change?

**Solution.**

**(a)** This is similar to Programming Assignment 2B, except that we don't use a regularizer and the learning rate is constant. We get pseudocode like the following:

```
Inputs: a list of example feature vectors $\boldsymbol{X}$
        a list of outputs $Y$
        learning rate $\eta$
$\boldsymbol{w} = (0, \ldots, 0)$
repeat
  select a training pair (input $\boldsymbol{x}$, output $y$)
  score = $\boldsymbol{w} \cdot \boldsymbol{x}$
  if $y$·score $\leq 0$
    $\boldsymbol{w} = \boldsymbol{w} + \eta \cdot y \cdot \boldsymbol{x}$
```

**(b)** This is the perceptron learning algorithm. (In the lectures, we didn't use a learning rate in the pseudocode, so if we set $\eta = 1$ we get exactly the same code as in the lectures.)

**(c)** Again, similar to our previous assignments and exercises. The solution will depend on how you introduce the regularizer. Generally, we will get something like the following pseudocode:

```
Inputs: a list of example feature vectors $\boldsymbol{X}$
        a list of outputs $Y$
        learning rate $\eta$
        regularizer weight $\lambda$
$\boldsymbol{w} = (0, \ldots, 0)$
repeat
  select a training pair (input $\boldsymbol{x}$, output $y$)
  score = $\boldsymbol{w} \cdot \boldsymbol{x}$
  if $y$·score $\leq 0$
    $\boldsymbol{w} = (1 - \eta \cdot \lambda) \cdot \boldsymbol{w} + \eta \cdot y \cdot \boldsymbol{x}$
  else
    $\boldsymbol{w} = (1 - \eta \cdot \lambda) \cdot \boldsymbol{w}$
```

## Question 11 of 12: Asymmetric machine learning tasks (9 points)

In some cases, machine learning problems are "asymmetric": mistakes are more critical in some circumstances than in others. Please discuss the following questions with respect to "asymmetric" machine learning tasks.

**(a, 1p)** Let's say we have a classification task with categories $A$, $B$, $C$, ... Can you think of an application where mistakes are more dangerous for some categories than others? For instance, it is more dangerous to classify a test case as $B$ if if the correct answer is $C$ than vice versa?

**(b, 2p)** How do you think this affects your evaluation protocol?

**(c, 2p)** How do you think this affects your training procedure?

**(d, 4p)** Please discuss regression tasks in a similar fashion as you did in (a)–(c).

**Solution.**

Different solutions are possible for all four questions here.

**(a)** In the simplest case, binary classification, we can imagine scenarios such as screening for some disease. In some cases, a false negative (missing a sick person) may be more undesirable than a false positive, or vice versa.

This example can be generalized, so that there might be more than one disease that we are looking for, where the costs of mistakes may vary between the different diseases.

**(b)** Assuming we can formalize the "importances" using some weighing scheme (e.g. class $A$ is twice as important as $B$), then we will typically use some sort of weighted evaluation metric in these scenarios.

For instance, the $F$-score is the harmonic mean of the precision and recall. Normally, precision and recall are equally important when computing the $F$-score, but in some cases we may want to make the evaluation more precision-oriented or recall-oriented. In scikit-learn, there is a function called `fbeta_score` where you can modify the importance of the recall.

More generally, if we have more than two classes, we might compute an average of the $F$-scores over the classes, weighted by the class importance scores.

Or in an even more detailed fashion, we may have costs associated with each cell in a confusion matrix.

In some use cases, we can quantify the (financial) cost directly in our application scenario, with respect to different types of mistakes. In such cases, it seems sensible to measure this cost directly.

**(c)** The notions of importances can be applied here too. For instance, we can design a weighted loss function that gives more importance to the training instances of some categories. A related approach is *oversampling* training instances in the more important categories: for instance, if category $A$ has a cost of 2 and category $B$ a cost of 1, we could duplicate each instance of the $A$ category.

Again, if we have different costs associated with each cell in a confusion matrix, we may design a loss function based on these costs.

**(d)** Again, there are some different ways that we can introduce similar ideas for regression tasks and you will have to clarify your ideas.

One approach could be that we say that it is more "dangerous" to over-predict than to under-predict, and again we could formalize this using evaluation protocols and loss functions that would have different weights for over-prediction and under-prediction. With MSE loss, we would get a "skewed" curve.

An alternative approach could be to say that the cost of mistakes depends on the true value. For instance, that for small $y$ values, we penalize deviations more than for large $y$ values. (But maybe in such applications, it would be more straightforward to log-transform the output variable?)

For example use cases, we can again think of medical applications, such as the blood pressure example in question 2. Maybe in this case we want to penalize the system more for under-predicting than for over-predicting?

## Question 12 of 12: Learning to play a game (8 points)

We would like to develop a machine learning system that plays a computer game: a game-playing "agent." There are different ways to train such agents, and in this task we will focus on training approaches where the system *learns from an expert*. This expert or teacher may be a human player or some other automatic game-playing system (e.g. a rule-based approach).

We will assume that this game allows a finite set of moves (such as *left*, *right*, *jump*, . . . ) and that it operates in discrete time: that is, it is a "step-by-step" game.

**(a, 2p)** Let's first assume that we want to "learn by watching": the expert plays a number of games, and we record all game states and what the expert did in each situation. Put formally, our data consists of a set of *demonstrations* of the game $D = d_1, \ldots, d_m$, and each demonstration $d_i$ is a sequence of game states and moves by the expert $(s_i^1, m_i^1), \ldots, (s_i^n, m_i^n)$.

How can we use these demonstrations to train a game-playing agent? Sketch an algorithm in pseudocode that describes how you would train the model.

**(b, 2p)** Now, let's change the training scenario so that instead of using the pre-recorded expert demonstrations, we assume that the expert is available "online" during training: for any situation in the game, we may ask the expert what is the best move under the current circumstances. Again, sketch an algorithm in pseudocode how you would train a game-playing agent.

**(c, 2p)** Do you see any advantages or disadvantages with the scenarios in (a) or (b)? Are they equivalent (meaning that they will generate the same training data) or not?

**(d, 2p)** Can you think of an application scenario that is not related to game-playing where these training approaches could be applied?

**Solution.**

This type of learning strategy is formally called *imitation learning*. Some of you might think of *reinforcement learning*, which is often discussed in the context of game playing: however, we have very explicitly framed this question in a way that makes it clear that reinforcement learning is not relevant here and RL-based solutions will get no points here.

**(a)** This seems like a fairly straightforward supervised setup: we just need to formalize a bit.

In the simplest case, we will say that the system we want to train is a standard supervised classifier that outputs game moves based on the current state. We would then generate a training set for this classifier simply by "flattening" the dataset: we put every state $s$ into the input part $X$ of the training data, and every corresponding expert move $m$ into the output part $Y$. We obviously need to have some sort of feature function $F(s)$ that computes a useful representation of a game state: for instance, in chess we would probably have a representation based on the pieces currently on the board and their relative positions. We finally train this classifier in the usual fashion. Written as pseudocode, we get:

```
def TrainAgent1:
    for each recorded demonstration $d_i$:
        for each state $s_i^j$ with expert move $m_i^j$:
            add $s_j^i$ to $X$ and $m_j^i$ to $Y$
    train a classifier on $X, Y$ and return it
```

In some games, it is probably going to be useful to generalize the approach described above by taking the history into account in some way. We can use a more complex feature function that considers not only the state at time $j$ but also the states at some previous time points; we may also consider the previous few moves. Recurrent neural networks (e.g. LSTM, GRU) can also be applied here: in this case, we would input some representation of the state (and the player's move) in each time step.

**(b)** This question requires a bit more thinking and you have more freedom in designing different possible solutions.

Let's start simple: again, we will generate a training set statically, then train the classifier. In this case, the pseudocode becomes something like this:

```
def TrainAgent2:
    for $i = 1, \ldots, m$:
        create an initial game state $s_i^1$
        while game not terminated:
            $m_i^j = \text{AskExpert}(s_i^j)$
            add $s_i^j$ to $X$ and $m_i^j$ to $Y$
            carry out the game move $m_i^j$ to get a new state $s_i^{j+1}$
            $j = j + 1$
    train a classifier on $X, Y$ and return it
```

But now we have more or less re-created what we had in (a): we have "recorded" demonstrations by the expert and used them as training data. Let's see if we can think of something different.

A problem with the type of supervision that we have seen until now is that the system will very rarely be exposed to problematic situations, if we assume that the expert plays the game well. This may lead to a *compounding* effect for errors: if the system makes a mistake, it is likely to end up in a situation that is different from what it has seen in the training data, which increases the chances of making more mistakes. It may be useful to try to create more variation in the training data by forcing the system "off the beaten path" in some way. The simplest way is probably to sometimes select a move randomly instead of asking the expert:

```
def TrainAgent3:
    for $i = 1, \ldots, m$:
        create an initial game state $s_i^1$
        while game not terminated:
            $m_i^j = \text{AskExpert}(s_i^j)$
            add $s_i^j$ to $X$ and $m_i^j$ to $Y$
            toss a coin that gives "heads" with probability $p$
            if "heads":
                $\hat{m}_i^j = m_i^j$
            else:
                $\hat{m}_i^j = \text{SelectRandomMove}$
            carry out the game move $\hat{m}_i^j$ to get a new state $s_i^{j+1}$
            $j = j + 1$
    train a classifier on $X, Y$ and return it
```

There are many ways that this approach can be modified. For instance, in the Dagger algorithm, which you can read about in this paper, we wouldn't select a random move, but instead use a previously trained classifier.

We can also imagine solutions where the expert is more tightly integrated into the learning process instead of generating the training data statically.

**(c)** The answer here depends on the type of game and on your solutions in (a) and (b).

If the game contains no random elements (e.g. in chess) and we use the first supervision strategy (TrainAgent2) described in (b), then we will get the same training data as in (a) (TrainAgent1). However, as we already discussed for (b), if we make sure that the system goes "off the beaten path" by sometimes not following the expert's advice (TrainAgent3), we might get a more robust system because we had more variation in the training data and the system will be more likely to recover from mistakes.

If we can play the game as long as we want, and the game can generate different "behaviors" (e.g. if it has a random behavior or random initial state), then then (b) allows us to generate as much training data as we think we need.

If it is time-consuming or financially expensive to ask the expert for advice (e.g. if the expert is human), then (a) seems more efficient than (b).

**(d)** We can imagine using similar approaches when training models for autonomous vehicles. Scenario (b) would correspond to a situation where we let the vehicle drive but a human driver is available to provide additional supervision. More generally, steering mechanisms for any type of machinery can probably be trained in this fashion as long as there are human operators available for supervision.

Imitation learning is also quite commonly used in some areas of natural language processing, in particular when developing some types of grammatical analysis tools (*parsers*). In NLP, the type of supervision we have in (a) is called a *static oracle* and (b) a *dynamic oracle*. Here is a link to a tutorial on imitation learning approaches in NLP.