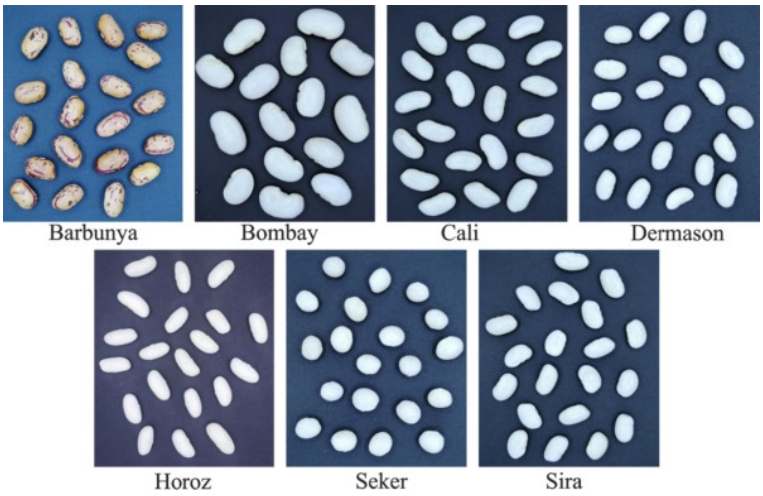
Solution to the exam
DIT866/DAT340: Applied Machine Learning, March 12–19, 2022

Question 1 of 12: Classifying dry beans (10 points)

We want to develop a system that classifies different varieties of dry beans, based on images. Our dataset contains beans of the following seven varieties:



For an image of a bean, we use image processing techniques to extract 16 different numerical features based on the bean's geometry such as the area, length of the major and minor axis, perimeter etc. We gather a dataset of about 13,000 beans; for each individual bean we include the name of the variety and extract the numerical features.

(a, 6p) Based on the dataset described above, explain how you would implement a machine learning model that can classify the image of a bean. You don't need to show Python code or give exact values of hyperparameters, but please give a description of the system and explain all steps you would carry out when developing it.

(b, 4p) The approach in (a) operates by extracting numerical features. Can you think of an alternative solution where we do not need this extraction step? Describe the approach briefly. You only need to mention the details that are different from your answer in (a).

Solution.

(a) This task is inspired by the paper *Multiclass classification of dry beans using computer vision and machine learning techniques* by Koklu and Ozkan (2020), and the image comes from their paper.

As usual, we start out the exam with an open-ended project-oriented question with many potential solutions. For a full score, you should cover the various aspects of the project discussed below in one way or another.

The *dataset* is given in this case, so we do not need to carry out any additional steps in collection or annotation. The instructions do not say anything about the balance of classes. If

there is significant imbalance, we may want to apply oversampling or undersampling to get a more balanced training set.

For experiments, we need to define a target *evaluation metric*. The accuracy probably makes sense in this case, since it is a classification task. Alternatively, precision and recall per class (in particular, if the classes are imbalanced). We reserve a test set for the experiments. In addition, we may optionally reserve a validation set for development; alternatively, we will use cross-validation during development.

We are using a feature-based approach, so it is probably a good idea to standardize/normalize the data and try to see if feature selection (e.g. `SelectKBest`) improves the quality while developing the system.

For the development of the classifier itself, we may apply a variety of learning algorithms and see which one works best, e.g. a tree ensemble such as `GradientBoostingClassifier` or `RandomForestClassifier`, or a neural network classifier (`MLPClassifier`).

Depending on what model we select, we will need to tune hyperparameters, e.g. the tree depth and ensemble size of random forests or the number of hidden units for neural networks. This tuning will ideally be done automatically (e.g. random search or grid search), which should be feasible in this case since the dataset is fairly small.

(b) Here, we do not want to use the numerical features, but classify a bean based on a raw image directly. So a natural solution here is to use a model for images, and in particular a CNN would be a natural solution.

The overall methodology is similar. In the project description above, the feature preprocessing and selection steps make less sense. It is probably still useful to normalize the dataset (e.g. so that the pixel values are between 0 and 1). Hyperparameter tuning is also generally more difficult for CNNs, and in practice we can typically just explore a small set of hyperparameter values. All other experimental settings would be the same as in (a).

For a basic CNN-based solution, we will typically use two-dimensional convolutional and pooling layers, followed by dense layers to compute the output. The last layer is a softmax with 7 outputs. When developing this system, it can be useful to consider regularization approaches including dropout, early stopping, and data augmentation (e.g. rotating the images or shifting the colors), in order to mitigate the problem of overfitting. To train the model efficiently, it is useful to consider different approaches to optimization (e.g. basic SGD vs Adam).

It is probably also a good idea to compare a basic CNN to one that uses a pre-trained model (e.g. based on ImageNet). It is generally difficult to know in advance whether this is going to be effective in a particular application, but it is always useful to try. In this approach, we use convolutional and pooling layers from an existing CNN and "plug" them into our CNN; we then train our output layers on top of this existing stack of layers.

Question 2 of 12: Feature selection (5 points)

Machine learning libraries such as scikit-learn often include some type of *feature selection*. For instance, in scikit-learn we may write

```
X, Y = ... our data ...

feature_selector = SelectKBest(k=10)
feature_selector.fit(X, Y)

X_reduced = feature_selector.transform(X)
```

(a, 3p) Explain informally what the code above does. In particular, describe how `SelectKBest` works.

(b, 2p) Can you describe some drawback of the technical approach used in `SelectKBest`? In particular, can you describe a situation where the most useful features are not included?

Solution.

(a) `SelectKBest` considers features one by one independently by measuring some sort of statistical association score between each feature and the output value `Y`. Generally, what we try to do here is to see how well we can predict Y by considering this feature alone: if we can predict Y almost perfectly then the score is high, and conversely if having access to the feature is almost as bad as having no feature at all, the score is low. Examples of such scores include the information gain (as we've seen in decision tree learning), and the ANOVA F-statistic (used by default in `SelectKBest`).

Then, we keep the 10 features with the highest association scores and discard the rest. If `X` is a matrix of shape (m, n), `X_reduced` will be a matrix of shape $(m, 10)$.

(b) This may happen when there are strong interactions between features. For instance, if X_1 and X_2 seem useless when considered individually, but where (X_1, X_2) is strongly predictive of Y.

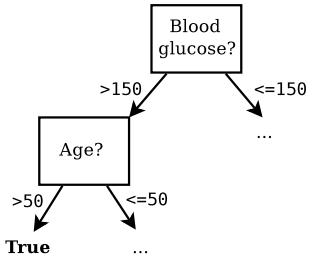
Conversely, since features are considered in isolation, it may happen that we select redundant features. For instance, if two features are identical, we might include both of them.

Question 3 of 12: Interpretable models? (4 points)

Why do you think some people claim that decision trees are more interpretable than neural networks? Please also explain and motivate your own opinion on this question.

Solution.

The question of interpretability is of course to some extent subjective, but the intuition is that in a decision tree, it is easy to understand what features have been involved in a decision. For instance, in the following decision tree



we can understand why the model has returned *True* if we know the blood glucose level and the the age.

With a neural network, we can explain the mechanics of the model: we computed the values hidden units, e.g. $h_1 = \text{activation}(\boldsymbol{w}_1\boldsymbol{x} + b_1)$, and then computed the output value as a function of the hidden units, e.g. $\text{output} = \text{sigmoid}(\boldsymbol{w}_o \cdot \boldsymbol{h} + b_o)$, but beyond the purely mechanistic explanation it is less intuitively obvious how the decision is affected by the values of the features.

The story becomes more nuanced when we have very large decision trees that are deep or uses a very large set of features. In addition, while it is easy to understand how a decision is made in a decision tree, it is less obvious to a user *why* the learning algorithm has decided to use these particular features.

In your solution, you will also have to reason about your own opinions here. We will accept anything that seems thoughtful.

Question 4 of 12: Predicting ejection fraction values (5 points)

At the Sahlgrenska hospital, we would like to develop a machine learning system that processes ultrasound images of the heart in patients that are investigated for cardiovascular diseases. The goal in our case is to develop a system that predicts the *ejection fraction* of the heart's left ventricle (LVEF). The LVEF is defined as the proportion of blood ejected from the left ventricle in one heartbeat. It is calculated as the stroke volume (the amount of ejected blood) divided by the end-diastolic volume (the ventricle's maximal volume).

(a, 1p) The following table shows the true LVEF values for 12 patients and the corresponding predictions by the system we have developed.

True LVEF	Predicted LVEF
0.36	0.55
0.56	0.56
0.71	0.68
0.63	0.77
0.32	0.36
0.48	0.39
0.34	0.51
0.35	0.31
0.65	0.64
0.39	0.34
0.45	0.54
0.29	0.45

Compute the mean absolute error of the predicted values for this dataset.

(b, 2p) The European Society of Cardiology guidelines refer to LVEF levels below 0.40 as *reduced*. For the dataset above, compute the precision and recall values for detecting patients with reduced LVEF.

(c, 2p) The Sahlgrenska hospital considers the development of this system to be a success and they want to sell it to other hospitals. However, the other hospitals then discover that when they measure the MAE values, they are considerably higher than what Sahlgrenska has previously reported. What are some possible explanations for this?

Solution.

(a) MAE = $\frac{1}{12}(|0.36 - 0.55| + \ldots + |0.29 - 0.45|) = 0.0842$

(b) P = 3/4 = 0.75; R = 3/6 = 0.50

(c)

- differences between cohorts (features of the patients are different, e.g. demographic factors; AND/OR distributions of LVEF values are different)

- differences between images, probably because of the equipment or how it is set up

Question 5 of 12: Clustering and classification (4 points)

Your company wants to develop a predictive system that outputs a discrete label for a given input. There is a large quantity of data available, but this data does not include the desired output labels, only the inputs.

Your colleague Astrid suggests that since there is no annotated data but a large amount of unannotated data, you should use an unsupervised clustering approach. Another colleague, Bertil, thinks that you should instead make an investment in data annotation and then use a supervised classification approach. Discuss the advantages and disadvantages of the two alternatives: when do you think Astrid's approach is preferable and when is Bertil's better?

Solution.

Intuitively, a clustering model should discover the most "natural" division of the dataset into groups – although this will depend on the feature representation. This division may or may not correspond to the prediction task we actually want to solve.

The advantage with Astrid's approach is of course that we don't need to invest in annotation, but in practice, in all but the easiest cases for clustering, the supervised approach will be better even with a small amount of data. Even if we choose to use clustering, we may need some annotated data in order to map clusters to the classes of the end task.

As a compromise, we may explore a semisupervised approach where we combine a small set of annotated data with a large amount of unannotated data.

We will probably see a range of answers here since it is a discussion, but for a full score here, the solution needs to discuss the challenge that the clustering model often does not come up with the distinction we are interested in from an application point of view.

Question 6 of 12: Tuning hyperparameters (4 points)

In the scikit-learn library, we can train *random forest* classifiers. For any machine learning model, the behavior of these classifiers will be influenced by various hyperparameters. For this type of classifier, the ensemble size (`n_estimators`) and the maximal tree depth (`max_depth`) are two important hyperparameters. For instance:

```
clf = RandomForestClassifier(n_estimators=100, max_depth=20)
```

(a, 2p) Describe the tradeoff you need to consider when tuning `n_estimators`. What happens if it is too low or too high?

(b, 2p) Describe the tradeoff you need to consider when tuning `max_depth`. What happens if it is too low or too high?

Solution.

(a)

- too low: unstable estimate, poor quality predictions

- too high: computationally heavy

(b)

- too low: underfitting

- too high: overfitting

Question 7 of 12: Classification of sounds (6 points)

We want to build a supervised classifier for sound recordings. This could, for instance, be a classifier that distinguishes different types of alarm signals.

We collect and annotate a large set of sound recordings. To make this discussion simpler, we

assume that all recordings are of the same length.

(a, 2p) Sketch a solution that uses a convolutional neural network.

(b, 2p) Sketch a solution that uses a recurrent neural network.

(c, 2p) Sketch a solution that uses neither of the architectures mentioned above.

The solutions should be brief but clear. Use pseudocode, formulas, or figures as you see fit.

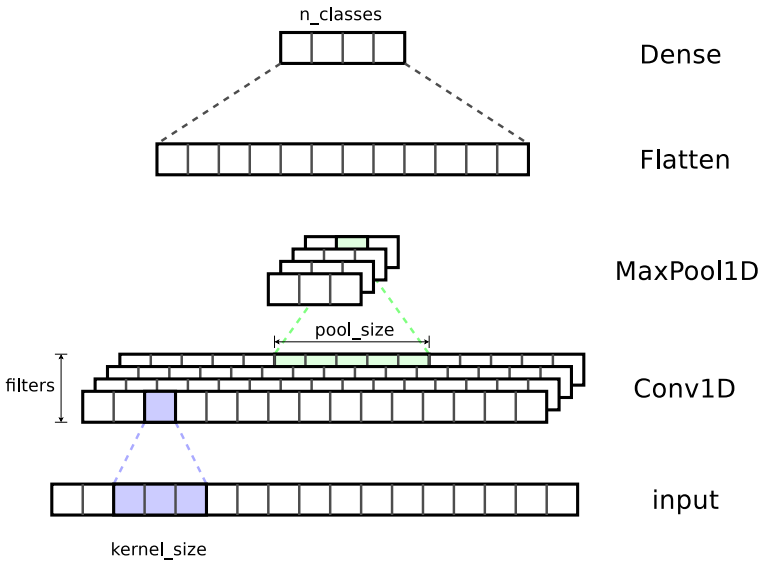
Solution.

In realistic ML models for sound, there are often preprocessing steps to transform the raw audio signal into some other representation, often based on a spectral representation (e.g. using a Fourier transform). However, we can also apply the ML model to the raw audio directly. Both types of representations are acceptable in (a)-(c), and we are expecting that most solutions will use just the raw audio.

(a)

The typical way to build a convolutional model would be to stack convolutional and pooling layers, and then "flatten" the representation and apply one or more dense layers to produce the final output.

The basic structure would be something like the following (using names of components and hyperparameters as in Keras):



For a full score, it needs to be mentioned explicitly in this context that we use *one-dimensional* convolutional and pooling layers, since we are working with a temporal signal, as opposed to images where we typically use 2D layers.

The hyperparameters will depend on how we set up the layers. For a solution as above, we need to specify the size of the convolutional filters (`kernel_size`), the number of filters (`filters`), the size of the pooling regions (`pool_size`). The number of output units will be

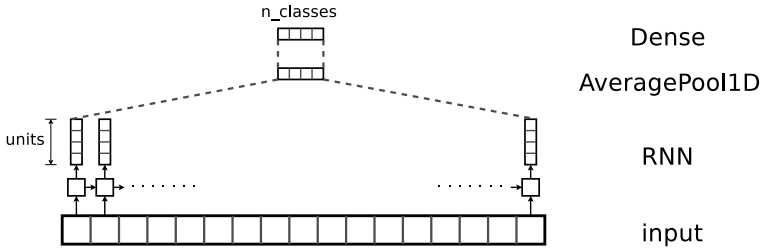
identical to the number of types of sounds.

(b)

When we say "RNN", in practice we will typically use either a LSTM or a GRU. With RNN-based solutions, we will have to connect to the output layer either by taking the last RNN state, or by some sort of pooling over all the states seen during the application of the RNN (typically the average). Here are figures visualizing these solutions.

Taking the last state:

Averaging the states:



Further alternatives include *attention* models that compute weighted averages. It is an empirical question which solution will perform better. Typically, for long sequences it can be more challenging to get the first solution to work well.

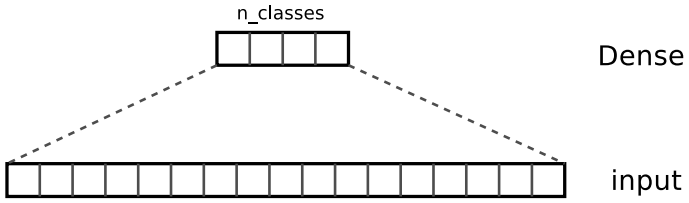
For a full score here, it is important here that it is made clear *how* the RNN connects to the output step.

For both types of solutions, we can optionally make the RNN bidirectional: that is, adding another RNN layer that is applied in the opposite direction. Furthermore, we have the option of adding additional RNN layers.

For the simple solutions above, the most important hyperparameter is the number of hidden units used by the RNN (`units`). Again, we need to specify the number of output classes in the final dense layer.

(c) Here, it can be expected that a wide variety of acceptable solutions will be proposed. My guess would be that they fall in two groups: (1) using some sort of informed feature extractor, followed by any type of supervised classifier; (2) applying a neural network (not a CNN or RNN) to the raw signal or a spectral representation. We will not discuss approach (1) because it would probably require quite a bit of signal processing that is out of scope here.

With approach (2), we can imagine several types of architectures. A minimal solution would be a regular feedforward model:



For all but the simplest sound classification tasks, this solution will probably not work particularly well unless we add some hidden layers. Again, the hyperparameters will depend on the selected architecture.

We accept anything here that is reasonably well-motivated and clearly explained. The architecture of the model needs to be clearly understandable and name-dropping is not sufficient (e.g. "I will use a transformer").

Question 8 of 12: Random forest classification (4 points)

The following pseudocode shows an incomplete implementation of a training algorithm for random forest classifiers. Modify the pseudocode so that you have a correct implementation.

def TrainRandomForest($\boldsymbol{X}, \boldsymbol{Y}, d_{\text{max}}, n$)
  for $i = 1, \ldots, n$
    $t_i \leftarrow$ TrainDecisionTreeClassifier($\boldsymbol{X}, \boldsymbol{Y}, d_{\text{max}}$)
  return the ensemble $\{t_i$ **for** $i = 1, \ldots, n\}$

def TrainDecisionTreeClassifier($\boldsymbol{X}$, $\boldsymbol{Y}$, d_{max})
  if all outputs in $\boldsymbol{Y}$ are identical **or** $d_{\text{max}} = 0$
    return a leaf with the majority class of $\boldsymbol{Y}$
  $F, T \leftarrow$ BestFeatureAndThreshold($\boldsymbol{X}, \boldsymbol{Y}$)
  $\boldsymbol{X}_{\text{low}}, \boldsymbol{Y}_{\text{low}} \leftarrow$ subset of training data where $F \leq T$
  $\text{tree}_{\text{low}} \leftarrow$ TrainDecisionTreeClassifier($\boldsymbol{X}_{\text{low}}, \boldsymbol{Y}_{\text{low}}, d_{\text{max}} - 1$)
  $\boldsymbol{X}_{\text{high}}, \boldsymbol{Y}_{\text{high}} \leftarrow$ subset of training data where $F > T$
  $\text{tree}_{\text{high}} \leftarrow$ TrainDecisionTreeClassifier($\boldsymbol{X}_{\text{high}}, \boldsymbol{Y}_{\text{high}}, d_{\text{max}} - 1$)
  return a tree where the top node splits on the feature F at the threshold T,
    and whose subtrees are tree_{low} and $\text{tree}_{\text{high}}$

The function BestFeatureAndThreshold is already implemented and does not need to be modified. The hyperparameters d_{max} and n correspond to `max_depth` and `n_estimators` in Question 6, respectively.

Solution.

The key points are (1) sampling a bootstrapped training set for each tree in the ensemble; (2) sampling a subset of the features each time we look for the best feature. Changes highlighted in blue. This solution uses one additional hyperparameter k, which is traditionally set to the square root of the number of features.

def TrainRandomForest($\boldsymbol{X}, \boldsymbol{Y}, d_{\text{max}}, n, k$)
  let m be the size of the training set

```
    for i = 1, ..., n
        bagging: X_i, Y_i ← sample m instances uniformly with replacement from X, Y
        t_i ← TrainDecisionTreeClassifier(X_i, Y_i, d_max, k)
    return the ensemble {t_i for i = 1, ..., n}

def TrainDecisionTreeClassifier(X, Y, d_max, k)
    if all outputs in Y are identical or d_max = 0
        return a leaf with the majority class of Y
    X_k ← sample k columns uniformly without replacement from X
    F, T ← BestFeatureAndThreshold(X_k, Y)
    X_low, Y_low ← subset of training data where F ≤ T
    tree_low ← TrainDecisionTreeClassifier(X_low, Y_low, d_max − 1, k)
    X_high, Y_high ← subset of training data where F > T
    tree_high ← TrainDecisionTreeClassifier(X_high, Y_high, d_max − 1, k)
    return a tree where the top node splits on the feature F at the threshold T,
        and whose subtrees are tree_low and tree_high
```

Question 9 of 12: Neural network regression (4 points)

We develop a neural network regression model using scikit-learn and train it on some dataset:

```
model = MLPRegressor(hidden_layer_sizes=(n,), activation='tanh')
```

In some industrial application, it is important to know about the behavior of the numerical output value from this model. For instance, the output value might be fed into some component that might be damaged if the value is too high.

Once we have trained the regression model, is there any way to find theoretical upper and lower bounds for the output values? That is, can you determine some values L and U so that we can guarantee that the model's output is always between L and U?

Your method should not need to run the model on any dataset and it should work for any number `n` of hidden units. The input might be very high-dimensional.

Solution.

It is easy to compute such an bounds if you remember that the output is a linear combination of `tanh` units: we can write the output as

$$\hat{y} = \sum_{i=1}^{n} w_o^i \tanh(\text{something}) + b_o$$

where w_o^i is the weight of hidden unit i in the output layer and b_o is the output bias term. Now, let's recall that the `tanh` function is bounded to the interval between `-1` and `1`. Let's say that the last layer has the weights `W_o` and the bias term `b_o`. The most natural upper bound is then `sum(abs(w) for w in W_o)+b_o`. We would get this output if all hidden units were "saturated" at either `+1` or `-1` (depending on whether a hidden unit's weight is positive or negative). This analysis does not take the interaction between hidden units into account, and it is possible that we are never close to this output value. Conversely for the lower bound: `-sum(abs(w) for w in W_o)+b_o`.

In scikit-learn, we can access the weights of the model via the attribute `coef_` and the bias terms as `intercepts_`.

Question 10 of 12: Training a linear classifier (3 points)

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

$$\text{Loss}(\boldsymbol{w}, \boldsymbol{x}, y) = \begin{cases} 1 & \text{if } y \cdot \boldsymbol{w} \cdot \boldsymbol{x} \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

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.

(a, 1p) Describe informally what we are trying to achieve by using this loss function.

(b, 2p) Assume that we want to train a classifier by minimizing this loss function on a training set. We are just using this loss and there is no regularizer. We implement the model in Keras and try to train it using the *Adam* optimizer. What will the result be?

Solution.

(a) This loss function is formally called the zero/one loss. We are counting misclassifications in the training set: *empirical risk minimization*. Equivalently, we are trying to maximize the classfication accuracy on the training set.

(b) This loss function is constant everywhere except at the point where it switches from 0 to 1. The subgradient will be 0 everywhere so gradient descent won't do anything at all: $\boldsymbol{w}$ will be stuck at the point of initialization.

Question 11 of 12: Collaborative filtering (7 points)

In a recommender system based on collaborative filtering, we build a model where we try to predict how well a user u would like an item i, e.g. a movie. Specifically, our model predicts a numerical rating $\hat{r}_{ui}$, such as a star rating between 0 and 5.

The model is defined as

$$\hat{r}_{ui} = \boldsymbol{p}_u \cdot \boldsymbol{q}_i + b_u + b_i$$

where $\boldsymbol{p}_u$ is a vector representing user u, $\boldsymbol{q}_i$ a vector representing item i, and b_u and b_i offset parameters.

(a, 4p) The training data consists of a set of user–item–rating triples. (For instance, "Astrid gives 4 stars to *Wild Strawberries*; Bertil gives 1 star to *Persona*.")

To train the model, we define a loss function and adjust the parameters ($\boldsymbol{p}_u$, $\boldsymbol{q}_i$, b_u, b_i for all users u and items i) to minimize this loss on the training data. In our case, we will use the squared error loss. For a training set D, we can write the total loss as

$$\text{Loss} = \sum_{(u,i,r) \in D} (\boldsymbol{p}_u \cdot \boldsymbol{q}_i + b_u + b_i - r)^2$$

For a single training instance u, i, r, the gradients of the loss with respect to the parameters are

$$\begin{aligned}
\nabla_{b_u}\text{Loss} &= 2 \cdot \delta_{ui} \\
\nabla_{b_i}\text{Loss} &= 2 \cdot \delta_{ui} \\
\nabla_{\boldsymbol{p}_u}\text{Loss} &= 2 \cdot \delta_{ui} \cdot \boldsymbol{q}_i \\
\nabla_{\boldsymbol{q}_i}\text{Loss} &= 2 \cdot \delta_{ui} \cdot \boldsymbol{p}_u
\end{aligned}$$

where δ_{ui} is the error $\boldsymbol{p}_u \cdot \boldsymbol{q}_i + b_u + b_i - r$.

Write the pseudocode for an algorithm to train the model. Make sure that you are explicit about what hyperparameters the user needs to provide.

(b, 1p) How would you use this model to recommend an item to a user?

(c, 2p) How applicable do you think this model is for recommending research papers to scientists?

Solution.

(a) This solution uses our standard SGD recipe. (Other options include *alternating least squares*, where we iteratively solve least squares problems until convergence.)

Inputs: a list of example feature vectors $\boldsymbol{X}$
a training set $D = (u, i, r)$
learning rate η
number of epochs N
embedding size m
initialize b_i and b_u to a random value for all u and i
initialize user embeddings $\boldsymbol{p}_u$ to a random vector of length m for all u
initialize item embeddings $\boldsymbol{q}_i$ to a random vector of length m for all i
repeat N times
  for each training instance u, i, r in D
    forward: compute the predicted value $\hat{r}$ according to the model
    backward: compute error $\hat{r} - r$ and the gradients according to the equations
    update:
      $b_u = b_u - \eta \cdot \nabla_{b_u}\text{Loss}$
      $b_i = b_i - \eta \cdot \nabla_{b_i}\text{Loss}$
      $\boldsymbol{p}_u = \boldsymbol{p}_u - \eta \cdot \nabla_{\boldsymbol{p}_u}\text{Loss}$
      $\boldsymbol{q}_i = \boldsymbol{q}_i - \eta \cdot \nabla_{\boldsymbol{q}_i}\text{Loss}$

(b) For a user u, apply the model to all items i such that have not been observed with u in the training set. Sort the items by the predicted rating and output a suitable subset.

(c) One well-known drawback of collaborative filtering is the *cold start problem*: that we have no information about new items (or new users). So if the purpose of this scientific paper recommendation system is for researchers to quickly find *new* papers they are interested in, this will probably be quite useless. In this case, a content-based approach (using e.g. a word-based representation) seems more applicable than collaborative filtering. On the other hand, if the collection of papers is fairly static, it could be more applicable.

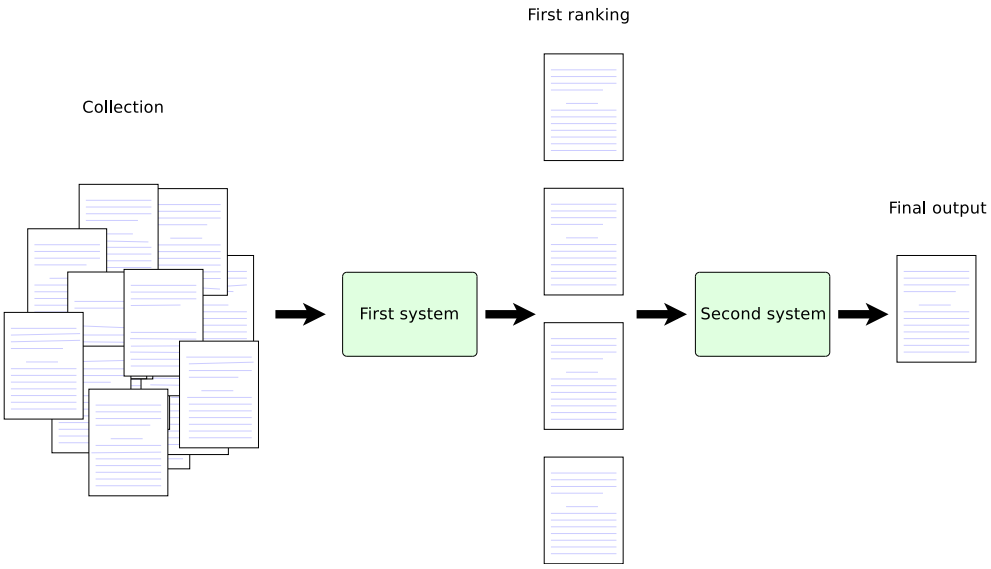
Question 12 of 12: Search (6 points)

In a search application, we want to build a system that finds the most relevant result given a query. This could, for instance, be a search over a very large collection of documents given some textual search query, or some sort of image search.

We want to implement this system using a two-stage process. In the first stage, we run a system that returns a list of up to N candididates, where N is a hyperparameter. This list includes a ranking computed automatically by the first system.

Next, we want to apply a second system to carry out the final selection of the best candidate.

The following figure shows the idea of the whole process.



(a, 1p) What do you think is the reason we want to implement the system using this two-stage approach?

(b, 5p) Describe how you would implement the second system.

You can assume that there is a fairly large set of queries for which the *gold-standard* ranking is available, in addition to the automatic ranking computed by the first system. That means that the output from the first stage is ranked so that the true best candidate result is ranked at the top. There may be ties in this ranking.

You can assume that there are existing feature functions f_q and f_c for queries and candidates, respectively.

Solution.

(a) I suppose there may be several reasons, but computational efficiency is probably the obvious one. Assuming that the collection is large, the first-stage search will have to be implemented very efficiently. In the second stage, we are less constrained by efficiency and this model can potentially be more "heavyweight" in terms of the machine learning model we use here.

(b) We need a machine learning solution here that allows us to pick the best candidate out of a set of up to N candidates. The input is given as a ranked list that potentially includes some ties – which means that occasionally, there might be more than one best candidate.

There are different ways to build a machine learning solution for this task. In the following, we will sketch a few different solutions. This is a non-exhaustive set of solutions, and you may be able to come up with some better approach.

Solution 1: Pairwise classification

In the first solution, we will consider the full ranking of the up to N candidates, even though we actually just care about the best candidate. We use a *pairwise* approach where for a given pair of candidates, we try to determine which of them is the best one. At runtime, we apply the model to all pairs, and select the output that gets the most "wins". This is in essence the same idea as one-versus-one approach to multiclass classification.

This would give us something like the following at training time:

Inputs: a training set D, where each instance consists of a query q_i and a list C_i of candidates
other hyperparameters...
$X \leftarrow$empty
$Y \leftarrow$empty
for each query q_i and candidate list C_i:
  for each candidate $c_j \in C_i$:
    for each candidate $c'_k \in C_i$:
      if c_j is better than c'_k:
        add (q_i, c_j, c'_k) to X and LEFT to Y
      else if c_j is worse than c'_k:
        add (q_i, c_j, c'_k) to X and RIGHT to Y
train a supervised binary classifier ϕ on the training set X, Y

The pairwise classifier would take a query–candidate–candidate tuple and apply the feature functions f_c once and f_q twice. Note that we have not included the ties here. This solution ignores the ranking produced by the first stage; this might have been added as an additional feature.

At runtime, we would have something like the following:

Inputs: a pairwise classifier ϕ, a query q, a candidate list C
initialize win counters for all candidates to 0
for each candidate $c_i \in C$:
  for each candidate $c'_j \in C \setminus \{c_i\}$:
    $r \leftarrow \phi(q, c_i, c'_j)$
    if $r =$ LEFT:
      increment win counter for c_i
    else:
      increment win counter for c'_j
return the candidate with the highest win count

We will need tie-breaking here if there are two candidates with the same number of wins. For

instance, in this case we may fall back on the first-stage ranking.

The drawback of the pairwise solution is that we need to apply the binary classifier ϕ up to $N \cdot (N-1)$ times, so this solution is probably not attractive if N is large.

Solution 2: Scoring the candidates

While the previous solution was inspired by one-versus-one multiclass classification, we may also adapt the one-versus-all approach to our task. Here, the approach is that we try to distinguish the best candidate(s) from the others.

Inputs: a training set D, where each instance consists of a query q_i and a list C_i of candidates
other hyperparameters...
$X \leftarrow$empty
$Y \leftarrow$empty
for each query q_i and candidate list C_i:
  for each candidate $c_j \in C_i$:
    if c_j is one of the best candidates:
      add (q_i, c_j) to X and TRUE to Y
    else:
      add (q_i, c_j) to X and FALSE to Y
train a supervised binary classifier ϕ on the training set X, Y

If there are ties for the top rank there will be more than one candidate that is added to the training set Y as TRUE.

At runtime, we simply return the top-scoring candidate. Note that this solution requires a classification model that can return some sort of prediction score and not only a binary decision.

Inputs: a ranking classifier ϕ, a query q, a candidate list C
for each candidate $c_i \in C$:
  $\text{score}_i \leftarrow \phi(q, c_i)$
return the candidate with the highest score

This solution seems computationally more efficient than the first one, since we only need to apply ϕ up to N times. However, we only consider each candidate in isolation here, while the first solution allows the model to take into account the relative merits of two candidates.

Solution 3: Adapting a learning algorithm

Finally, we may also create a new learning algorithm that works directly with this task.

The following is an example of this approach, where we adapt the perceptron learning algorithm. The idea that we try to realize here is that we want the best candidates to be ranked above the other candidates. And the training approach is the usual one in the perceptron: if we make a mistake, then we update the model. This algorithm uses a prediction step that is identical to what we used in Solution 2.

Inputs: a training set D, where each instance consists of a query q_i and a list C_i of candidates

number of epochs T
$\boldsymbol{w} \leftarrow$all zeros
for each epoch $1, \ldots, T$:
  for each query q_i and candidate list C_i:
    let $\hat{c}$ be the predicted candidate, based on the current $\boldsymbol{w}$
    if $\hat{c}$ is not one of the best candidates:
      let c be one of the best candidates
      $\boldsymbol{w} \leftarrow \boldsymbol{w} + f_c(c) - f_c(\hat{c})$

Computationally, this has the same properties as Solution 2. The perceptron-based solution may require that we add some additional features, since currently the query features f_q are not used at all.