Data Science Machine Learning Question Bank

Practical Questions and solutions that will strengthen your understanding and prepare you for tough interviews

Venkatesh K

Willy

1 Practical Machine Learning Problems

1.1 Predicting party affiliation

We would like to build a system that tries to predict which candidate an American voter will prefer in the 2020 presidential election: Republican, Democratic, another party, or abstaining. This system has access to extensive information about each voter, from which we can construct the features that the classifier will be using. The system should train a classifier, and then evaluate that classifier on a separate test set.

(a) Sketch an implementation of the system in Python. You should use standard functions in scikit-learn as far as possible. If you don't remember the names of scikit-learn functions and classes, just use pseudocode or invented names, as long as it is clear what you mean. (It's OK if you exclude the imports.)

You can propose any feature that you think could be useful for this task, except features related to voting, which is what we are trying to predict, or completely unrealistic features such as reading the voter's mind. You may assume that you have access to all the resources (e.g. databases of personal, financial, and geographical data) that you need to compute the features that you have defined, and that there are Python functions to deal with that data. Your implementation needs to use at least three different features.

(b) Discuss briefly the ethical implications of building and using such a classifier.

Solution.
(a) The idea of this task is twofold: thinking of what features could be useful for the task, and showing that you can use the scikit-learn library. Here's a typical solution. Obviously, all the different `look_up` functions are invented, and my solution assumes that we have access to a collection of people identified by their social security numbers. The important thing here is that the solution needs to include some credible features Then, ideally the code is reasonably close to something similar to scikit-learn.

```
from sklearn.feature_extraction import DictVectorizer
from sklearn.svm import LinearSVC
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score
from sklearn.cross_validation import train_test_split

def extract_features(soc_sec_nbr):
    x = {}
    x['gender'] = look_up_gender(soc_sec_nbr)
    x['race'] = look_up_race(soc_sec_nbr)
    x['home_state'] = look_up_race(soc_sec_nbr)
    x['education'] = look_up_education(soc_sec_nbr)
    x['age']  = look_up_age(soc_sec_nbr)
    x['income']  = look_up_income(soc_sec_nbr)
    return x
```

```
if __name__ == '__main__':
    soc_sec_nbrs = ... # we assume that we can access a selection of voters

    # create features for all instances
    X = [ extract_features(nbr) for nbr in soc_sec_nbrs ]

    # we make the unrealistic assumption that we know the vote of each
    # person -- let's say this might come from interviews
    Y = [ look_up_vote(nbr) for nbr in soc_sec_nbrs ]

    # split the data into training and test parts
    Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y,
                                                    train_size=0.8,
                                                    random_state=0)

    # build a pipeline consisting of a vectorizer
    # and a learning algorithm
    classifier = make_pipeline(DictVectorizer(),
                               GradientBoostingClassifier())

    # train the model
    classifier.fit(Xtrain, Ytrain)

    # predict
    Yguess = classifier.predict(Xtest)

    # evaluate
    print(accuracy_score(Ytest, Yguess))
```

(b) Obviously, as soon as we are dealing with personal data, we need to be careful. In particular, if we'd like to use voting preferences to train a classifier, it seems reasonable to ask for the consent of the people who are included in the training set.

In general, it is good to be careful when a classifier is categorizing *people*. For many people, it may be sensitive to be labeled by such a classifier ("you seem like a typical X voter").

This is an open-ended question with no clear-cut answer. A passable answer needs to demonstrate a certain awareness that we can't apply these kinds of methods tools blindly.

1.2 Distinguishing between varieties of Arabic]

The regional varieties of spoken Arabic are very diverse: for instance, someone from the Gulf would normally have great difficulties understanding the colloquial speech of a Moroccan. Moreover, the colloquial varieties are all quite different from the formal, standardized register: Modern Standard Arabic. While the standard register is used in formal writing, the regional varieties tend to dominate not only in the spoken language but also in many types of informal written text, such as in the social media.

Our annotators have collected a fairly large set of YouTube comments written in the Arabic script, and categorized each of them by its language variety. Texts written in standardized Arabic are tagged *MSA*. The regional varieties are divided into five broad groups: *Egyptian*, *Gulf* (including the Arabian Peninsula), *Iraqi* (and Kuwait), *Levantine* (from Syria, Jordan, Lebanon, and Palestine), and *Maghrebi* (North African). We also include a category *Other* for documents written in the Arabic script in languages other than Arabic, such as Persian or Urdu, so in total we have seven categories. Here are two examples:

بونجوغ عليكم لميسيو نتاع هاد السيمانة والو ماعجبتنيش قاع	`Maghrebi`
جان الشعب من مؤيدي هاي الرئيس	`Gulf`

(a) Sketch an implementation in Python of a system that automatically classifies a short text in the Arabic script into one of the seven categories mentioned above. The system should train a classifier, and then evaluate that classifier on a separate test set. You should use standard functions in scikit-learn (or another machine learning toolkit) as far as possible. If you don't remember the names of scikit-learn functions and classes, just use pseudocode or invented names, as long as it is clear what you mean.[1]

There is a Python function that reads the annotated dataset:

```
texts, labels = read_arabic_dataset()
```

This function returns two equally sized lists of strings: the document texts, and the corresponding language categories.

Make sure that your explanation is clear about what features the system uses. It is up to you to decide what features to use. The features should not be based on any external resources: they should just use what's available in the documents.

(b) How do you think we should analyze the errors of this classifier?

Solution.

This is a type of document classification problem, so we can apply the "cookie-cutter" solution for this type of problem: representing the documents as bag-of-words feature vectors, which are then used to train some classifier. Here is the Python code of a typical solution. This becomes very compact since we're relying on the built-in tokenizer of `CountVectorizer`. (You're not required to remember all the import statements.)

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.svm import LinearSVC
from sklearn.pipeline import Pipeline
from sklearn.cross_validation import train_test_split
from sklearn.metrics import accuracy_score

if __name__ == '__main__':
    docs, labels = read_arabic_dataset()
    X_train, X_test, Y_train, Y_test = train_test_split(docs, labels,
                                                        train_size=0.5,
                                                        random_state=0)

    cls = Pipeline([('vec', CountVectorizer()),
                    ('cls', LinearSVC())])
```

[1]This task is inspired by the Master's thesis by Wafia Adouane, *Automatic Detection of Under-resourced Languages: Dialectal Arabic Short Texts*, University of Gothenburg 2016, and the examples are taken from her text.

```
    cls.fit(X_train, Y_train)
    Y_guess = cls.predict(X_test)
    print(accuracy_score(Y_test, Y_guess))
```

However, it's quite likely that a word-based feature representation isn't optimal for discriminating between languages. Probably it's better to build a representation based on character N-grams. Here is a solution that extracts N-grams with N from 2 to 5. Again, the solution is short because the vectorizer has a built-in N-gram counting functionality, but extracting N-grams can of course also be done manually. Note also that we're using a `TfidfVectorizer` instead of the `CountVectorizer`, in order to downweight features that are common in all categories.

```
from sklearn.feature_extraction.text import TfidfVectorizer
...

if __name__ == '__main__':
    docs, labels = read_arabic_dataset()
    X_train, X_test, Y_train, Y_test = train_test_split(docs, labels,
                                                        train_size=0.5,
                                                        random_state=0)
    cls = Pipeline([('vec', TfidfVectorizer(analyzer='char',
                                            ngram_range=(2, 5))),
                    ('cls', LinearSVC())])

    cls.fit(X_train, Y_train)
    Y_guess = cls.predict(X_test)
    print(accuracy_score(Y_test, Y_guess))
```

Other possible solutions include character-based recurrent neural networks, but this can't be done inside scikit-learn at the moment, so your solution would need to be described using another toolkit or with pseudocode.

(b) Obviously, trying to explain the errors is easier if we actually know the languages. Otherwise, it may be meaningful to look at the distributions of errors. Which classes are most commonly confused with each other? [In the Master's thesis that inspired this question, the error analysis showed that languages of neighboring regions were more easily confused, e.g. there were more Moroccan/Algerian mistakes than Moroccan/Gulf.]

1.3 Bug fixing

A large software development company receives bug reports from the users of their various software products. Each bug report concerns one particular product, and the report itself consists of a description of the problem, typically one or a few paragraphs of free text. There is some additional information, such as the severity level (on a scale from 1 to 5), the platform (e.g. Mac or Windows) where the problem occurred, and optionally some text that describes how to reproduce the problem. The company then assigns the bug to a software developer, who is responsible for coming up with a solution.

(a) The company would like to develop a machine learning system that tries to estimate how much time will be spent to fix the problem. How would you design such a system? What type of model would you try to use? What features would it need?

(b) How would you suggest that the system should be evaluated?

Solution.
(a) A fairly typical regression problem. We can probably use all the information mentioned in the description directly, including the text represented as a bag of words (or TF-IDF). It seems likely that it is important to include the information about the software developer: either just the name, or any other information we might have about that person (e.g. education, experience).

Using scikit-learn, we can probably use any of the typical regression models.

```
regressor = make_pipeline(DictVectorizer(), StandardScaler(), Ridge())
```

If the problem turns out to be nonlinear and a linear regressor doesn't work, we can try a nonlinear model, e.g. a neural network regressor (`MLPRegressor`) or a tree-based ensemble (`GradientBoostingRegressor`, `RandomForestRegressor`).

(b) For the prediction problem itself, the typical choice would be a standard regression metric (e.g. MSE, R^2).

1.4 Chinese word segmentation [

In East Asian scripts including the Chinese script, words are normally not separated by whitespace. This makes it more difficult than in European scripts to automatically segment a sentence string into separate words, which is a prerequisite for most types of automatic analysis of the text. For instance, the following sentence[2]

這正是往日屯門海岸線之標誌。

can be written as separate words as follows:

這 正是 往日 屯門 海岸 線 之 標誌 。

(a) Describe how you would address the problem of automatic Chinese word segmentation using machine learning techniques. Program code isn't mandatory, but you can use pseudocode or Python if necessary for your explanation. Your solution should explain what types of learning techniques and features you would use, and how they would be applied. As a training set, the system is allowed to use text: either raw text, or text that has been segmented manually. No other resource is allowed (including dictionaries).[3]

(b) How do you think such a system should be evaluated?
Solution.
(a) This question is intentionally open-ended, because its purpose is to test your skills in taking a problem and casting it as a machine learning task. So there are many imaginable ways in which this task could be solved, and any clearly described solution that is credible and well motivated will get a full score. The following is a typical solution.

Example solution: supervised sequence tagging. The most straightforward solution is probably to treat this as a supervised learning task using sequence tagging techniques. To convert the task

[2]I have copied this example from an external source, but I've been told a couple of times by Chinese speakers that these characters are weird. Feel free to provide me with a better example! You get the point anyway.
[3]Inspired by a paper by Nianwen Xue, *Chinese Word Segmentation as Character Tagging*, 2003.

into sequence tagging, we need to mark the word boundaries in some way. Then the example sentence would be tagged

```
B B I B I B I B I B B B I B
```

After this conversion, it's just a matter of applying some effective sequence model. The easiest solution is to use a greedy step-by-step classifier, using a set of features describing the current character, surrounding character, etc.

Or we could use a specialized learning algorithm for sequence problems (e.g. conditional random field), or a sequence-based neural network (e.g. LSTM) using embeddings of the characters.

(b) This seems like a typical situation for a precision/recall evaluation:

$$P = \frac{\text{number of correctly detected words}}{\text{number of proposed words}}$$

$$R = \frac{\text{number of correctly detected words}}{\text{number of words in gold standard}}$$

Computing an accuracy of the predicted beginning/inside tags would also be a way to evaluate the system.

2 Dealing with Features

2.1 Selecting a good set of features]

When using machine learning techniques, it is crucial that we provide the learning system with useful *features*. Here is an example of a set of 28 features used in a system that classifies a sentence by difficulty for a language learner. (The six difficulty levels are defined according to the European standard: A1–C2.)

Nr.	Feature Name	Feature ID
	Traditional	
1	Sentence length	SentLen
2	Average token length	TokLen
3	Percentage of words longer than 6 characters	LongW%
4	Type-token ratio	TTR
	Syntactic	
5	Average dependency depth	DepDepth
6	Dependency arcs deeper than 4	DeepDep
7	Deepest dependency / sentence length	DDep / SentLen
8	Ratio of right dependency arcs	RightDep
9	Modifiers	Mod
10	Modifier variation	ModVar
11	Subordinates	Sub
12	Prepositional complements	PrepComp
	Lexical-morphological	
13	Average word frequency (Wikipedia list)	WikiFr
14	Average word frequency (Kelly list)	KellyFr
15	Percentage of words above B1 level	DiffW%
16	Number of words above B1 level	DiffWs
17	Percentage of words at B1 level	B1W%
18	Lexical density	LexD
19	Nouns/verbs	NN/VB
20	Adverb variation	AdvVar
21	Modal verbs / verbs	MVB/VB
22	Participles / verbs	PCVB/VB
23	S-verbs / verbs	S-VB/VB
24	Percentage of S-verbs	S-VB%
25	Relative pronouns	RelPN
	Semantic	
26	Nominal ratio	NomR
27	Pronoun/noun	PN/NN
28	Average number of senses per word	Sense/W

When developing a classifier based on these features, it is important to understand that the best possible classifier does not necessarily use all 28 of them. In general it can be harmful for a classifier to add useless features, because they can "drown out" the good features.[4]

(a) Why can't we write a practical algorithm that automatically determines a perfect set of features?

(b) Describe some automatic or manual method to determine a good set of features.

(c) We'd like to train a neural network that uses all the 28 features listed in the table. We don't need to go into the details about the features, but we can note that all of them are numerical. Is there anything in particular that you need to think of when training the classifier?

Solution.
(a) As already mentioned, the best feature set isn't necessarily the largest, so we need to find the best subset of features. And since the features interact in nontrivial ways, the only sure way to find the best set is by trying out all possible subsets on some development set. Now, how many subsets are there? In general, if we have N features, there are 2^N possible subsets (because each feature is either included or not). For small feature sets this is feasible, but for instance with Ildikó's features, we have 2^{28} possible subsets, or about 268 million. So we'd need to try out 268 million different classifiers. In practice, we have to use some rule of thumb to find a fairly good feature set, which isn't necessarily the best one. This leads us to ...
(b) If we start with a manual approach, we could carry out an *ablation test* (which was an optional task in the first assignment). This means that for each feature F, we evaluate a classifier that uses *all* features, and another classifier that uses all features except F. We then remove all features that don't seem to cause a drop in performance when removed.

This approach can of course be automatized in various ways. In *greedy backward selection*, we start with the full set, try to find the feature that gives the highest improvement by being

[4]The features were described in the Master's thesis by Ildikó Pilán, *NLP-based approaches to sentence readability for second language learning purposes*, University of Gothenburg, 2013.

removed, and then repeat again with the reduced set. Conversely, *greedy forward selection* starts with an empty feature set and gradually adds features until there is no feature that gives an improvement.

Another alternative that is more efficient, but possibly less precise, is to use some sort of statistical feature scoring function. For instance, the `SelectKBest` class in scikit-learn will rank the features by some measure of statistical "relatedness" between each feature and the output variable. (See `http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html`.) We could then select a subset by selecting e.g. the top 10 features according to this ranking.

You just have to explain one approach, which can be either manual or automatic.

(c) These features are all numerical. Neural networks (as well as linear classifiers) can be affected negatively if the scales of the features are very different. (For instance, if the first feature has a mean of about 20, and the fourth feature about 0.5.) The instructions didn't say anything about the scales of the features, but in general with neural networks and this type of features, it's good to standardize the features. So a `StandardScaler` or a `MinMaxScaler` in the pipeline is probably a good idea.

2.2 General feature selection questions

What are *filter*, *wrapper*, and *embedded* feature selection methods?

Solution.
Filter methods. These methods select features before training, typically by ranking features according to some criterion of "usefulness," based on how strongly each single feature correlates with the variable we are trying to predict. Given such a ranking of features, we can select e.g. the top 1000 features, as in `SelectKBest` in scikit-learn. An examples of feature ranking functions is the mutual information score, also known as information gain; this is called `mutual_info_classif` in scikit-learn.

See also Section 1.3 in this description of decision trees. The feature ranking functions used in decision tree learning can be used generally, not only when training decision trees.

Wrapper methods. These methods use a machine learning method as a part of the selection process, by training the model based on different subsets of the features and then trying to find the subset that gives the best performance. This is difficult to do exactly, so often *greedy* approximations are used, such as *forward* or *backward* greedy feature selection.

Embedded methods. This refers to machine learning methods that perform feature selection as a part of their training process. One example of such a method is decision tree learning where the number of branches is limited, because then only the most "useful" features will be included. Another example is the use of the L_1 or *Lasso* regularizer in linear models: for mathematical reasons, when using this regularizer we often get an optimum where several of the feature weights are set to exactly zero, which means that those features can be removed entirely. The number of features removed will depend on the regularization tradeoff hyperparameter.

3 Decision Trees

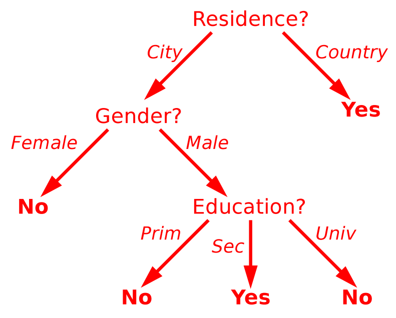
3.1 Building a decision tree

The folllowing dataset can be used to train a classifier that determines whether a given person is likely to own a car or not. There are three features: education level (primary, secondary, or university); residence (city or country); gender (female, male).

education	residence	gender	has car?
sec	country	female	yes
univ	country	female	yes
prim	city	male	no
univ	city	male	no
sec	city	female	no
sec	country	male	yes
prim	country	female	yes
univ	country	male	yes
sec	city	male	yes
prim	city	female	no
univ	city	female	no
prim	country	male	yes

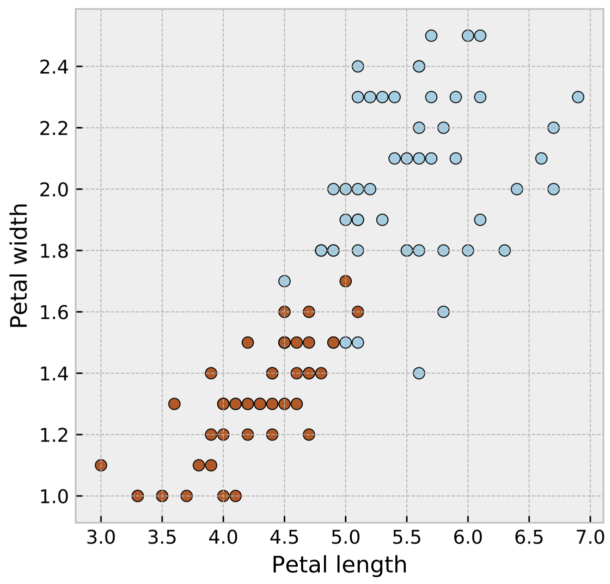
Make a decision tree that classifies the training set perfectly. You can construct it manually or use an automatic algorithm.

Solution.
Here is one possible decision tree.

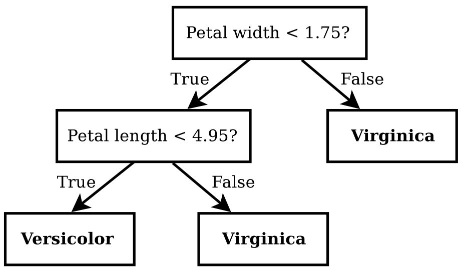


3.2 Classifying irises

One of the most widely known datasets in machine learning is the *Iris* data, which was originally collected in 1936 by Ronald Fisher. The following scatterplot shows petal measurements (width and height) for two subspecies, *Iris versicolor* (red) and *Iris virginica* (blue).

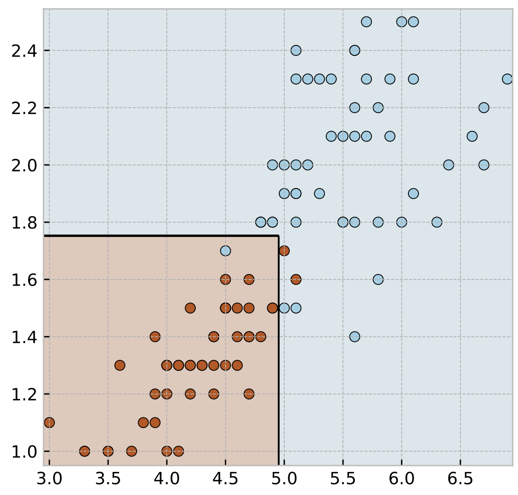


We train scikit-learn's `DecisionTreeClassifier` to distinguish between the two types of irises. Here is the result:



Draw the classifier's decision boundary.

Solution.
This is just about reading the tree. Here is the decision boundary:



3.3 General decision tree questions

(a) Describe the differences between how *discrete-valued* and *numerical* features are handled by decision tree learning algorithms.

Solution.
For a description of how numerical features are handled, see Section 2.3 in this document about decision trees.

(b) Describe the differences between how decision tree models are used for *classification* and for *regression*.

Solution.
For a description of decision tree regression, see Section 3 in this document about decision trees.

(c) Mention one or more approaches to mitigating the problem of *overfitting* in decision tree models.

Solution.
A non-exhaustive list of possible approaches:

- Limiting the maximally allowed depth of the tree,
- limiting the maximally allowed number of splits,
- "tree pruning" by removing tree branches that seem to have no effect when evaluated on a validation set.

- 5: Decision tree classifiers

- 5: Decision tree classifiers

4 Linear Classification

4.1 The perceptron algorithm

(a) Write down the perceptron algorithm for training binary classifiers. You can write it in pseudocode or Python (either using standard data structures, or NumPy).

(b) Let's assume that we'd like to develop a sentiment polarity classifier for review texts. We have the following small training set:

```
X = [ ['this', 'movie', 'was', 'good'],
      ['this', 'movie', 'was', 'really', 'bad'],
      ['it', 'is', 'as', 'good', 'as', 'its', 'predecessor'],
      ['it', 'is', 'so', 'bad', 'that', 'I', 'have', 'no', 'words'] ]
Y = ['positive', 'negative', 'positive', 'negative']
```

What does your perceptron classifier contain after two iterations through this training set? That is, what numbers are stored in the data structure that you use to represent the classifier?

Solution.
(a) The perceptron learning algorithm can be written in a variety of different ways (and all would be counted as correct), but here we will stick to the version presented in the lectures. Here's the pseudocode, using vector notation. (For solutions in Python, please take a look at the code from the second lecture.)

```
Inputs: a list of example feature vectors $X$
        a list of outputs $Y$
        the number of iterations $N$
$w$ = zero vector
repeat $N$ times
  for each training pair $(x_i, y_i)$
    score = $w \cdot x_i$
    if $score \leq 0$ and $y_i$ belongs to the positive class
      $w = w + x_i$
    if $score \geq 0$ and $y_i$ belongs to the negative class
      $w = w - x_i$
```

(b) We assume that the documents are represented using an unweighted bag-of-words feature representation. Then, after looking at the first two instances, the weight vector (or weight dictionary) stores the number +1 at the dimension corresponding to the word *good* and -1 for *really* and *bad*. This weight vector will not change as we look at the two other examples, and it will also remain unchanged during the second iteration over the training set.

4.2 Breakfast habits [ ]

We would like to build a classifier that will try to predict whether a given person will prefer tea or coffee for breakfast. For each person, we have features representing his or her nationality and occupation. Here is the Python code:

```
from sklearn.feature_extraction import DictVectorizer
from sklearn.linear_model import Perceptron
from sklearn.pipeline import make_pipeline

X = [ { 'nationality':'british', 'occupation':'student' },
      { 'nationality':'swedish', 'occupation':'student' },
      { 'nationality':'british', 'occupation':'lawyer' },
      { 'nationality':'swedish', 'occupation':'chef' } ]
Y = ['tea', 'coffee', 'tea', 'coffee']
p = make_pipeline( DictVectorizer(),
                   Perceptron(n_iter=1) )
p.fit(X, Y)
```

(a) Explain what the `DictVectorizer` is and what it will do when we call `p.fit(X, Y)`.
(b) Explain in detail what the `Perceptron` will do when we call `p.fit(X, Y)`. Use pseudocode or Python to explain how the algorithm works. For simplicity, you can assume that there are just two classes in `Y`.
(c) What output we will get if we run the following code, and why do we get that output?

```
new = { 'nationality':'british', 'occupation':'chef' }
print(p.predict(new))
```

Solution.
(a) The `DictVectorizer` is responsible for converting features into numerical vectors. The `Dict` is because each instance is represented as a dictionary, which is a natural choice when we talk of attributes that have values.

When we call `fit`, it will first gather a vocabulary of observed attribute–value pairs. Each of them will be assigned its own dimension in a vector space. So after calling fit, there will be some feature-to-dimension mapping, maybe something like this:

`nationality:british`	0
`nationality:swedish`	1
`occupation:chef`	2
`occupation:lawyer`	3
`occupation:student`	4

After creating this mapping, it will transform the training instances into vectors. Assuming that we have the mapping above, the training set becomes the following matrix:

$$\begin{array}{ccccc} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{array}$$

(b) When we call `fit`, the `Perceptron` will train a classifier using the perceptron learning algorithm. Its input is the vectorized feature matrix, as output by the `DictVectorizer` described in (a).

The perceptron learning algorithm can be written in a variety of different ways (and all would be counted as correct), but here we will stick to the version presented in the lectures. Here's the pseudocode, using vector notation. (For solutions in Python, please take a look at the code from the second lecture.)

```
**Inputs:** a list of example feature vectors $X$
            a list of outputs $Y$
assign one output class as the "positive" class and the other one as the "negative"
$w$ = zero vector
```

```
repeat n_iter times
  for each training pair $(\boldsymbol{x}_i, y_i)$
    score = $\boldsymbol{w} \cdot \boldsymbol{x}_i$
    if $score \leq 0$ and $y_i$ belongs to the positive class
      $\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{x}_i$
    if $score \geq 0$ and $y_i$ belongs to the negative class
      $\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{x}_i$
```

As stated in the task, we assume that the classification problem is binary. To handle more than two classes, we could use a multiclass perceptron or convert the multiclass problem into several binary problems, using the `OneVsRestClassifier` for instance.
(c) Here, you need to say at least what weights the perceptron has learned (which will depend on how you formulate the algorithm), and what score will be computed for this instance. You don't need to go through all steps, but here's a walkthrough anyway.

`coffee` $\rightarrow$ positive, `tea` $\rightarrow$ negative
Step 1: $\boldsymbol{w} \cdot \boldsymbol{x}_1 = 0 \Rightarrow \boldsymbol{w}$ becomes -1 for British (dimension 0) and for student (4)
Step 2: $\boldsymbol{w} \cdot \boldsymbol{x}_2 = -1 \Rightarrow \boldsymbol{w}$ becomes -1 for British (0) and +1 for Swedish (1)
Step 3: $\boldsymbol{w} \cdot \boldsymbol{x}_3 = -1$, no change
Step 4: $\boldsymbol{w} \cdot \boldsymbol{x}_4 = +1$, no change

So the British chef, unlike the Swedish chef, will be classified as a tea drinker ($\boldsymbol{w} \cdot \boldsymbol{x} = -1$).

4.3 Understanding logistic regression [recycled exam question]

The weight vector $\boldsymbol{w}$ in a logistic regression classifier is defined as the $\boldsymbol{w}$ that minimizes the function f in the following equation:

$$f(\boldsymbol{w}, \boldsymbol{X}, Y) = \frac{1}{N} \sum_{i=1}^{n} L(\boldsymbol{w}, \boldsymbol{x}_i, y_i) + \frac{\lambda}{2} \cdot R(\boldsymbol{w})$$

As usual, $\boldsymbol{X}$ is a list of feature vectors of all the instances in the training set and Y the corresponding outputs (coded as +1 or -1), and N the size of the training set. The notation $\sum_{i=1}^{n}$ means that we sum over all instances in the training set. The parameter λ (Greek letter *lambda*) is discussed below.

Specifically, the functions L and R are defined as follows:

$$L(\boldsymbol{w}, \boldsymbol{x}_i, y_i) = \log(1 + \exp(-y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i)))$$

and

$$R(\boldsymbol{w}) = \|\boldsymbol{w}\|^2$$

which is the squared vector length of $\boldsymbol{w}$, defined as $\sum_{j=1}^{m} w_j^2$ where m is the number of dimensions in the vector (and w_j the j:th element of $\boldsymbol{w}$).

(a) What is the purpose of $L(\boldsymbol{w}, \boldsymbol{x}_i, y_i)$ and $R(\boldsymbol{w})$, respectively?

(b) What happens to $\boldsymbol{w}$ if λ is set to a high value such as 1000000? What if it is a low value such as 0.000001?

(c) In the second lab assignment, the parameter η (Greek letter *eta*) was not an input to the algorithm but was instead set automatically in a way so that it decreased gradually during training. Why is this often better than using a constant value for η?

Solution.
(a) L is the *loss function*, which shows how well the classifier fits the training set. In this case, L is the log loss because we are training a logistic regression classifier. It is the log of the output of the sigmoid function, which represents the probability assigned by our model to the correct output. R is the *regularizer*, which represents the "simplicity" of the classifier in some way – the Occam's razor intuition. The regularizer used here will penalize large weights.

(b) λ controls the tradeoff between the loss and the regularizer: between fitting the training data and keeping the classifier simple. If λ is a large number, the weights will become very small values. Conversely, if it's a small number, the classifier will "work harder" to fit the training set, so some weights will probably be quite large.

(c) If η is too large, our step length is too long, so we may have difficulties to find the optimum exactly because we are "jumping over" it. On the other hand, if it's too small, then we may be moving too slowly in the beginning. (See the lecture on optimization for a visual explanation.)

4.4 More logistic regression questions

(a) We have trained a logistic regression classifier and the first weight in its weight vector is 5. What does this tell us?

(b) Can you think of a practical advantage of using a logistic regression classifier, as opposed to e.g. a perceptron or support vector classifier, in terms of interpretability of the results?

(c) How can we interpret the output of a two-class logistic regression classifier as a probability?

(d) In a two-class logistic regression model, the weight vector $\boldsymbol{w} = [4, 3, 2, 1, 0]$. We apply it to some object that we'd like to classify; the vectorized feature representation of this object is $\boldsymbol{x} = [-2, 0, -3, 0.5, 3]$. What is the probability, according to the model, that this instance belongs to the positive class? (You should be able to solve this without a calculator.)

(e) Discuss one or more solutions for building a logistic regression model that can handle more than two output classes.

(f) We have a classification problem where our feature representation contains about 10,000,000 features. We'd like to develop a classifier that can be deployed in a mobile phone, so preferably it should have a small memory footprint. Discuss one or more solutions for how this can be done.

Solution.
(a) It shows that the feature associated with the first dimension has a positive effect on the classifier's decision function: if we increase the value of this feature, the classifier increases its probability that this instance belongs to the positive class. More precisely: if we increase the feature's value by one, we increase the classifier's output score value by 5.

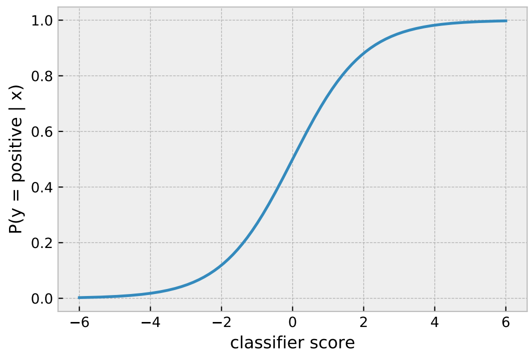
[Mathematical side note: the score $\boldsymbol{w} \cdot \boldsymbol{x}$ in logistic regression can be interpreted as the *logarithm of the odds*: see `https://en.wikipedia.org/wiki/Logit`.]

(b) LR gives a probabilistic output (if we apply the sigmoid function to the classifier's score), which makes it a bit easier to interpret the output. "Probability of 90%" is probably easier to understand than "output score of 2.20".

(c) As already discussed in (b): by applying the sigmoid (logistic) function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

A plot of the sigmoid function:



(d) The classifier's output score $w \cdot x = -13.5$. A large negative score, so the probability will be practically zero.
(e) Let's assume that we have three classes: A, B, and C.

(alternative 1) Convert the 3-class problem into separate binary problems. The easiest approach is *one-vs-rest*, where we have one binary classifier dedicate to each class. For instance, one classifier that looks for the class A, etc. Then when we classify an instance, we output the class that is associated with the binary classifier that gives the highest output score.

(alternative 2) Use a *softmax* model instead of a sigmoid.

$$P(A) = \frac{e^{\text{score}(A)}}{e^{\text{score}(A)} + e^{\text{score}(B)} + e^{\text{score}(C)}}$$

(f) One solution (which is not specific to logistic regression) is to apply a feature selection method. (For instance `SelectKBest` in scikit-learn.) Then we can just decide how many features we'd like to include, and we can easily control the memory/accuracy tradeoff.

A second solution would be to use a logistic regression model that uses L_1 regularization (similar to a *Lasso* linear regression model.) When using this type of regularizer (and setting the regularization parameter so that the regularizer is strong), most of the weights in the weight vector will be set to zero, which means that we can ignore these features.

4.5 Understanding support vector classifiers [ ]

The weight vector $\boldsymbol{w}$ in a linear support vector classifier is defined as the $\boldsymbol{w}$ that minimizes the function f in the following equation:

$$f(\boldsymbol{w}, \mathbf{X}, Y) = \frac{1}{N} \sum_{i=1}^{n} L(\boldsymbol{w}, \boldsymbol{x}_i, y_i) + \frac{\lambda}{2} \cdot R(\boldsymbol{w})$$

As usual, $\boldsymbol{X}$ is a list of feature vectors $\boldsymbol{x}_i$ of all the instances in the training set and Y the corresponding outputs y_i (each output coded as +1 or -1), and N the size of the training set. The notation $\sum_{i=1}^{n}$ means that we sum over all instances in the training set. λ is a parameter set by the user.

Specifically, the functions L and R are defined as follows:

$$L(\boldsymbol{w}, \boldsymbol{x}_i, y_i) = \max(0, 1 - y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i))$$

and

$$R(\boldsymbol{w}) = \|\boldsymbol{w}\|^2$$

which is the squared vector length of $\boldsymbol{w}$, defined as $\sum_{j=1}^{m} w_j^2$ where m is the number of dimensions in the vector (and w_j the j:th element of $\boldsymbol{w}$).

(a) Let's assume that the weight vector $\boldsymbol{w} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ and the feature vector $\boldsymbol{x} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$, and $y = -1$. What's the output of the function $L(\boldsymbol{w}, \boldsymbol{x}, y)$ in this case? What does this tell us?

(b) What's the purpose of the function R?

(c) The equation above tells us what kind of $\boldsymbol{w}$ we'd like to have, but it doesn't really tell us how to get it. Explain on a high level how to convert the equation into an actual algorithm. (It isn't mandatory to use pseudocode or equations to answer this question.)

Solution.
(a) L is called the *hinge loss* function. As usual, a loss function is designed to output a low value when an instance is classified correctly; more specifically, the hinge loss reaches its minimum value (zero) if the instance has a margin of at least 1 to the decision boundary. In this case, $\boldsymbol{w} \cdot \boldsymbol{x} = -2$ and $y = -1$, so we get $L(\boldsymbol{w}, \boldsymbol{x}, y) = \max(0, 1 - 2) = 0$, so this instance is classified well enough: the loss is 0.
(b) R is the *regularizer*, which represents the "simplicity" of the classifier in some way – the Occam's razor intuition. The regularizer used here will penalize large weights.
(c) To convert the equation into a training algorithm, we need to apply an optimization algorithm to the objective function. In the course, we have discussed *stochastic gradient descent* (SGD) algorithm repeatedly, so this will be our natural choice. In SGD, we train in a step-by-step fashion, by randomly selecting one training example in each step. We compute the gradient (more precisely, the subgradient) of the objective f with respect to that example, and update the weight vector: $\boldsymbol{w} = \boldsymbol{w} - \eta \cdot$ gradient, where η is a step length parameter. In this case, the gradient of the regularizer plus the loss is

$$\lambda \cdot \boldsymbol{w} - \begin{cases} y_i \cdot \boldsymbol{x}_i & \text{if } y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i) < 1 \\ \text{zero vector} & \text{otherwise} \end{cases}$$

where $\lambda \cdot \boldsymbol{w}$ is the gradient of the regularizer, and the rest is the gradient of the hinge loss.

To get a full score here, you need to mention at least that we're using an optimization algorithm such as SGD, and something about the update step in SGD being related to the gradient of the function f; as stated above, you don't need to write down the formulas.