Solution to the example exam
LT2306: Machine learning, October 2016

Score required for a VG: 22 points

Question 1 of 6: Hillary or the Donald? (6 points)

We would like to build a system that tries to predict which candidate an American voter will prefer in the 2016 presidential election: Clinton, Trump, another candidate, 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, 5p) 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, 1p) 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. To get 5 points, a solution needs to include some credible features, and code that's reasonably close to scikit-learn. (It's OK if you exclude the imports.)

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

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

    # predict
    Yguess = classifier.predict(Xtest)

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

Actually, this pipeline will probably not work fantastically in practice, since we're mixing numerical features (of different magnitudes) with symbolic features. It's probably good to include a `Normalizer` in the pipeline, so that the numerical features (in particular the income) are scaled down.

(b) Obviously, as soon as we are dealing with personal data, we need to be very 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.

Also, it's difficult to see a use case for this classifier that isn't a bit dubious, maybe except as a statistical analysis, for determining correlation of features with voting patterns. Probably many people prefer not to be labeled ("you seem like a Trump voter"). For an example of classification gone wrong, see `https://twitter.com/jackyalcine/status/615329515909156865`.

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 – you don't need to include all the discussions I wrote here.

Question 2 of 6: The perceptron algorithm (6 points)

(a, 4p) 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, 2p) 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 $\boldsymbol{X}$
        a list of outputs $Y$
        the number of iterations $N$
$\boldsymbol{w}$ = zero vector
repeat $N$ 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
      $\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{x}_i$
```

(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.

Question 3 of 6: Understanding logistic regression (6 points)

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:

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

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). 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, 1p) What is the purpose of $L(\boldsymbol{w}, \boldsymbol{x}_i, y_i)$ and $R(\boldsymbol{w})$, respectively?
(b, 1p) 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, 3p) Below is the pseudocode of a training algorithm that (approximately) solves the optimization problem in the logistic regression model. Explain on a high level how the algorithm is derived from the equation in the optimization problem.

```
**Inputs:** a list of example feature vectors $\boldsymbol{X}$
            a list of outputs $Y$
            the parameter $\lambda$
            the number of steps $T$
            step length $\eta$
$\boldsymbol{w}$ = zero vector
**repeat** $T$ times
    select a random training pair $(\boldsymbol{x}_i, y_i)$
    score = $\boldsymbol{w} \cdot \boldsymbol{x}_i$
    $\boldsymbol{w} = (1 - \eta \cdot \lambda) \cdot \boldsymbol{w} + (\eta \cdot \frac{y_i}{1+\exp(y_i \cdot \text{score})}) \cdot \boldsymbol{x}_i$
```

(d, 1p) 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) To convert the equation into a training algorithm, we need to apply an optimization algorithm to the objective function. In this case, we apply the *stochastic gradient descent* (SGD) algorithm. In SGD, we train in a step-by-step fashion, by selecting one training example in each step. We compute the gradient of the objective f with respect to that example, and update the weight vector: $\boldsymbol{w} = \boldsymbol{w} - \eta \cdot$ gradient. In this case, the gradient of the regularizer plus the loss is

$$\lambda \cdot \boldsymbol{w} - \frac{y_i}{1 + \exp(y_i \cdot \text{score})} \cdot \boldsymbol{x}_i$$

To get 3 points here, you need to mention at least that we're using stochastic gradient descent, and something about the update step being related to the gradient of the function f.

(d) 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.)

Question 4 of 6: Neural network classification (6 points)

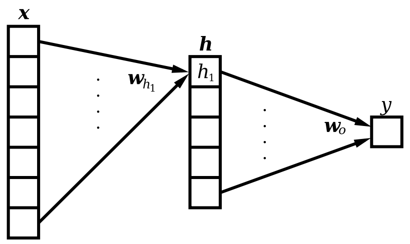
(a, 4p) Describe how classification is done in a binary feedforward neural network classifier with one hidden layer. Explain how the input looks, and the steps that are carried out by the classifier until the output is produced.
(b, 2p) Describe in general terms how a feedforward neural network can be trained.

Solution.
(a) As usual, the input to the classifier is a numerical vector $\boldsymbol{x}$ encoding our features. This is more or less like in the other classifier (SVM, LR), and $\boldsymbol{x}$ could be prepared using one of the vectorizers in scikit-learn. A difference in practice is that $\boldsymbol{x}$ tends to have a lower dimensionality when working with NNs, and in particular for words we prefer to use *embeddings* rather than one-hot encoding.

The first step in a feedforward NN is to compute the values in the *hidden layer*, which will result in a new vector $\boldsymbol{h}$. Each number in $\boldsymbol{h}$ is the result of applying a "sub-classifier": for instance, to compute the first number in $\boldsymbol{h}$, let's call it h_1, we use a weight vector $\boldsymbol{w}_{h_1}$ specifically dedicated to h_1. We compute the score by using the dot product between this weight vector and the input vector $\boldsymbol{x}$, and then applying an *activation function* a_h. For instance, the sigmoid is a common choice of activation function, and in this case the "sub-classifier" is like a logistic regression classifier.

When all the numbers in $\boldsymbol{h}$ have been computed in this way, we apply a similar procedure to compute the output. But now, we use $\boldsymbol{h}$ instead of the input vector $\boldsymbol{x}$.

Here is a figure illustrating this process:



Alternatively, using matrix notation, we can explain the NN classification in a less verbose way. We make the vector $\boldsymbol{h}$ of hidden-layer outputs by first multiplying the weight matrix $\boldsymbol{W}_h$ with the input vector $\boldsymbol{x}$, and then applying the activation function a_h to the result of the matrix multiplication. (The activation function is applied element by element in the vector.)

$$\boldsymbol{h} = a_h(\boldsymbol{W}_h \cdot \boldsymbol{x})$$

And the output value is computed is a similar fashion, but using the hidden-layer vector instead:

$$y = a_o(\boldsymbol{W}_o \cdot \boldsymbol{h})$$

To answer this question and get a full score, you can draw a figure, use formulas, or just explain in plain text. A complete solution will mention that the input is a numerical vector, and how each “layer” is computed from a previous layer (weighted score and activation function).

(b) Just like for logistic regression in Question 3, it’s about defining an objective function that we optimize. This will typically be a loss function (for instance the log loss) applied to the network’s output. Most commonly, we’ll then apply some sort of stochastic gradient descent, as in the logistic regression training algorithm. The gradients of the weights are computed using the *backpropagation* algorithm, which works by first computing the value of the output and all hidden units, and then computing the gradients in each layer backwards from the output through the lower layers. A passable solution to this question will at least mention that we’re doing optimization based on gradients.

Question 5 of 6: Troublesome words (6 points)

Features based on *words* are used in many machine learning systems for language processing.
(a, 3p) Since word vocabularies are large, we may run into problems because some word features are observed in the training data just a few times or not at all. Describe at least one method to make machine learning systems based on word features more robust.
(b, 3p) We train a classifier for polarity classification of movie reviews, based on a bag-of-words feature representation. Explain why this classifier is likely to have a lower classification accuracy if applied to reviews of toys, and suggest at least one idea for dealing with this problem.
Solution.
(a) We need some way to generalize word-based features, so that we aren’t completely helpless when we encounter some word that we haven’t seen before. The most common way nowadays is to use a large corpus to train word clusters (such as Brown clusters) or embeddings. With clusters, we group words which behave similarly into a cluster, so that there could be a cluster consisting of person names or locations, for instance. Similarly, word embeddings place closely related words near each other in a vector space.

As an alternative to building word representations from a corpus, we could try to use a lexical resource such as WordNet.
(b) The general problem is that the distribution of features differs between domains, and our learning algorithms don’t know which features are generally useful and which are just useful in some domain. This is particularly true of word features, which of course can differ a lot beween domains. For instance, if we train a classifier on movie reviews, the classifier will probably pick up some movie-related words, such as *fast-paced* or *well-directed*, which are less relevant for classifying reviews of toys. And conversely, there might be toy-related words (for instance *sturdy*) that are predictive of some polarity but that our classifier doesn’t know about, because we don’t generally see them in movie reviews.

To address this problem, we can try to apply any of the *domain adaptation* techniques that we saw in the last lecture. For instance, if we have a lot of unlabeled toy reviews, we can try to build word clusters or embeddings using the toy review corpus as in (a). Or we could try instance weighting: giving a higher importance to those training examples in the movie corpus that are more similar to reviews in the toy corpus.

Question 6 of 6: Finding speech events (6 points)

We would like to develop a system that finds *speech events* in text. For instance, in the following sentence

“It’s a problem that clearly has to be resolved,” said David Cooke, executive director of the RTC.

the system should spot that there's a speech event because of the word *said*, which we refer to as a *trigger word*. In addition to the triggers, we'd like to find the *message* and *speaker* of each speech event; for instance, in the example sentence, the message is *It's a problem that clearly has to be resolved* and the speaker *David Cooke, executive director of the RTC*.

(a, 5p) Describe how you would address this problem using machine learning techniques. Program code isn't mandatory, but you can use pseudocode or Python if it's needed for your explanation. Your solution should explain what types of learning techniques you would use, how they would be used, and what kind of information (features, preprocessing, external resources) you would need. Also, if you make any simplifying assumption to make the problem more tractable, please describe them.

(b, 1p) 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 solution that is credible and well motivated and comes reasonably close will get a full score. I will give a few example solutions here, just to show clearly that they can be completely different.

Solution 1: sequence tagging. We could view this task as a markup task, which as we have seen in the third assignment can be solved using sequence tagging techniques. To convert the task into sequence tagging, it's best to make the parts of the markup (trigger, speaker, message) into word-by-word tags, for instance IOB (inside/outside/beginning) tags, as in the third assignment. Here's an example:

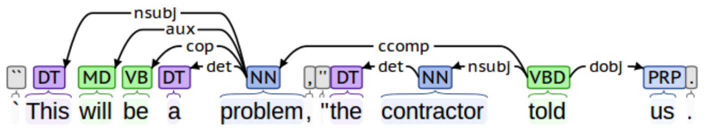
```
‘‘ Remember Pinocchio ?     ’’ says   a     female voice .
O  B-msg    I-msg     I-msg O  B-trg  B-spk I-spk  I-spk O
```

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 as in the assignment (and optionally use beam search or Viterbi at test time). Or we could use a specialized learning algorithm for sequence problems (e.g. structured perceptron or conditional random field), or a sequence-based neural network (e.g. LSTM).

If we're using a step-by-step classifier as in the assignment, we need to think of the features. For this type of model, features tend to be extracted from a small window around the current token under consideration. For instance, it's probably useful to include a previous token feature when finding the start of the message, since they are often preceded by a quotation mark. In addition to token features, we might think of using the output of a part-of-speech tagger or maybe even a named entity recognizer (which might help us to find the speakers). For a sequence-based NN, we typically don't use any features except the word embedding of the current token.

Limitations: this is an easy and appealing solution, and probably more efficient than the two solutions below. There are a couple of limitations. First, that it can't detect speech events inside other speech events – but these are probably quite rare anyway. For instance: *Upon returning to New York, "Exhausted, I got into a taxicab, and the woman driver said: 'Americans make better fishermen,' " he recalled.* Another limitation of this solution is that if the sequence tagger finds a message or a speaker, it doesn't tell us which speech event (that is, which trigger) it is connected to. And the model is free to propose speakers and messages without even finding a trigger. This can probably be taken care of if we detect the triggers in a first step, and then find the speakers and messages.

Solution 2: classifying nodes in a parse tree. Another solution would be to apply a parser to the sentence. We could then use a two-stage process: first we have a classifier that spots the triggers, and then a second classifier that finds the speaker and message by looking at *subtrees* in the parse tree. For instance, here is a sentence containing a speech event, and as you can see, the speaker and message are subtrees directly connected to the trigger in this case.



The most important feature for the first classifier is obviously the word itself and possibly other words around it: this subtask is similar to Solution 1. For the second classifier, it seems natural to include a feature that shows how a subtree is related to the trigger, e.g. `nsubj` for the speaker and `ccomp` in the example above. But other features could also be useful: maybe again something to pick up the quotation marks.

Limitations: we need to have a parser, and the success of our system is dependent on the parser being accurate. English parsers are quite good nowadays, but if we'd like to work with another language, this approach might work less well. This solution does *not* have the limitation of Solution 1: it's can work even if speech events are nested. On the other hand, it will not work across sentence boundaries, unless we add some special solution to handle this case. For instance: *Jane told me "Tarzan is my man. He is the king of the jungle."*

Solution 3: using a semantic role labeler for preprocessing. I think this solution is a bit boring (and overkill), since semantic role labeling (SRL) can be seen as a general task of which finding speech events is a special case. SRL systems find events of different types, and extracts their participants. For instance, if using an SRL system based on FrameNet, we can look for events belonging to the frame STATEMENT and then just pick out the SPEAKER and MESSAGE as well. Not much machine learning necessary here, and as I mentioned, this makes the task a bit too easy... *Limitations:* this is an expensive solution because we need to have an existing SRL system at hand, and general SRL is quite difficult. (Our special case is probably easier to do reliably.) Also, it limits the applicability in languages beside English, where SRL systems of decent quality are rare. And as in the parse-tree-based solution, we are probably limited to finding statements contained within single sentences.

(b) Again, this is quite an open-ended question and the design of the evaluation protocol will reflect what we think is important. Anyway, I think that the natural approach will probably be to use a labeled precision and recall evaluation: something quite similar to the evaluation we used in the third assignment when evaluating the named-entity recognizer.

To do this, we count the number of correctly detected triggers: that is, the number of triggers in your output that exactly match coincide with some trigger in the gold standard. We then divide this number by the number of predicted triggers to get the precision, and by the number of triggers in the gold standard to get the recall.

We then compute precision and recall values for the detection of speakers and of messages. This evaluation should ideally take into account that a speaker or message needs to be correctly linked to a trigger in order to be counted as correctly detected.

Alternatively (if we're using Solution 1) we could compute the accuracy on the level of IOB tags, but I think this is a bit less interesting.