Artificial Intelligence	**Assignment 6**
comp440/comp557 Fall 2017	**6 November 2017**

This assignment is due November 24 at 8 pm on Canvas. Download `assignment6.zip` *from Canvas. There are five problems worth a total of 145 regular points + 20 extra credit points for comp440 and 165 regular points for comp557. Problems 1 and 5 require written work only, Problems 2, 3 and 4 require Python code and a writeup. All written work should be placed in a file called* `writeup.pdf` *with problem numbers clearly identified. All code should be included at the labeled points in* `submission.py` *in the folders* `ner`, `text_classification` *and* `image_classification`. *For Problems 3, 4 and 5 please run the autograder using the command line* `python grader.py` *and report the results in your writeup. Upload all code directories and* `writeup.pdf` *on Canvas by the due date/time.*

1 Decision networks (10 points)

Consider a student who has the choice to buy or not buy a textbook for a course. We will model this decision problem with one boolean decision node B, indicating whether the student chooses to buy the book, and two Boolean nodes M, indicating whether the student has mastered the material in the book, and P indicating whether the student passes the course. There is a utility node U in the network. A certain student, Sam, has an additive utility function: 0 for not buying the book and -$100 for buying it; and $2000 for passing the course and 0 for not passing it. Sam's conditional probability estimates are:

$$\begin{array}{ll} P(p|b,m) = 0.9 & P(p|b,\neg m) = 0.5 \\ P(p|\neg b,m) = 0.8 & P(p|\neg b,\neg m) = 0.3 \\ P(m|b) = 0.9 & P(m|\neg b) = 0.7 \end{array}$$

You might think that P would be independent of B given M. But, this course has an open-book final – so having the book helps.

- (4 points) Draw the decision network for this problem.
- (5 points) Compute the expected utility of buying the book and not buying the book.
- (1 points) What is the optimal decision for Sam?

2 Conditional random fields and named entity recognition (40 points + 20 EC points for comp440/required points for comp557)

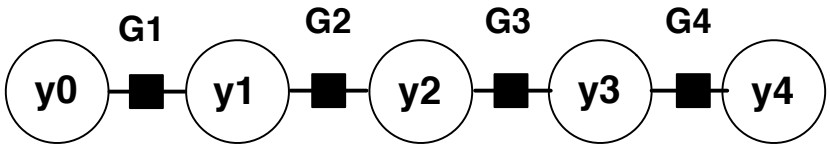
One of the principal aims of natural language processing is to build a system that can automatically read a piece of text and determine who is doing what to whom. A first step towards that goal is named-entity recognition, the task of taking a piece of text and tagging each word as either person, organization, location, or none of the above. Here is an example.

```
        --PER--            --LOC--
In 1971, Obama returned to Honolulu to live with his maternal grandparents,
 --PER--   --PER--  --PER--
Madelyn and Stanley Dunham, and with the aid of a scholarship attended
 --ORG-- --ORG--
Punahou School, a private college preparatory school, from fifth grade until
his graduation from high school in 1979.
```

In this assignment, we will build a named-entity recognition system using factor graphs representing conditional random fields (CRF). We will start with a chain-structured factor graph called a linear-chain conditional random field, which admits exact inference via variable elimination. Then, for extra credit for comp440, and required for comp557, we will develop a more sophisticated factor graph to capture long-range dependencies which are common in natural language. For this model, we will use Gibbs sampling to perform approximate inference. All of the code is in the folder `ner` and you must place your code in `submission.py` inside `ner`.

Linear-chain conditional random fields

Let $\mathbf{x} = (x_1, \ldots, x_T)$ be a sequence of words and let $\mathbf{y} = (y_1, \ldots, y_T)$ be a sequence of tags. We will model the Named Entity Recognition (NER) task with the following factor graph. Here the



tags are the variables, and the words $\mathbf{x}$ only affect the potentials G_t (denoted by the black boxes). The probability of a tag sequence $\mathbf{y}$ given $\mathbf{x}$ is

$$
\begin{aligned}
p(\mathbf{y}|\mathbf{x};\theta) &= \frac{1}{Z(\mathbf{x};\theta)} \prod_{t=1}^{T} G_t(y_{t-1}, y_t; \mathbf{x}, \theta) \\
Z(\mathbf{x};\theta) &= \sum_{\mathbf{y}} \prod_{t=1}^{T} G_t(y_{t-1}, y_t; \mathbf{x}, \theta)
\end{aligned}
$$

where y_0 = `-BEGIN-`, a special tag indicating the beginning of a sentence, and $Z(\mathbf{x};\theta)$ is the normalization constant. The potentials G_t are

$$G_t(y_{t-1}, y_t; \mathbf{x}, \theta) = exp(\theta.\phi_{local}(t, y_{t-1}, y_t, \mathbf{x}))$$

where ϕ_{local} is the local feature function and $\theta \in \Re^d$ is the parameter vector. $\theta.\phi_{local}(t, y_{t-1}, y_t, \mathbf{x})$ stands for the dot product of the parameter vector θ with the feature vector ϕ_{local}. ϕ_{local} can depend arbitrarily on the input $\mathbf{x}$, and will generally access the words around position t (i.e., x_{t-1}, x_t, x_{t+1}).

We have provided you with the function `LinearChainCRF.G(t,y_,y,xs)` to compute the value of $G_t(y_{t-1}, y_t; \mathbf{x}, \theta)$, where `y_` is y_{t-1}, `y` is y_t and `xs` is $\mathbf{x}$. In mathematics, indexing starts from 1, so

y_1 is the first tag, but in Python, indexing starts from 0 (i.e., `ys[0]` is the first tag). To get the value of $G_3(y_2, y_3; \mathbf{x}, \theta)$, call `LinearChainCRF.G(2,ys[1],ys[2],xs)` where `ys` is the tag sequence $\mathbf{y}$ and `xs` is the observation sequence $\mathbf{x}$. For y_0 =`-BEGIN-`, use the provided constant `BEGIN_TAG`. For example, $G_1(-$`BEGIN`$-, y_1; \mathbf{x}, theta)$ is `LinearChainCRF.G(0,BEGIN_TAG,ys[0],xs)`.

Problem 2.1: Inference in linear chain CRFs (30 points)

- (10 points) Our first task is to compute the best tag sequence $\mathbf{y}*$ given a sentence $\mathbf{x}$ and a fixed parameter vector θ.

$$\begin{aligned} \mathbf{y}* &= argmax_{\mathbf{y}} P(\mathbf{y}|\mathbf{x}; \theta) \\ &= argmax_{\mathbf{y}} \prod_{t=1}^{T} G_t(y_{t-1}, y_t; \mathbf{x}, \theta) \end{aligned}$$

  The Viterbi algorithm eliminates the variables $y_1, \ldots, y_T$ in the order y_1 to y_T, producing a sequence of forward messages:

$$\begin{aligned} Viterbi_1(y_1) &= G_1(-\texttt{BEGIN}-, y_1; \mathbf{x}, \theta) \\ Viterbi_2(y_2) &= max_{y_1} Viterbi_1(y_1) G_2(y_1, y_2; \mathbf{x}, \theta) \end{aligned}$$

  Repeating this process gives us the following algorithm:

  1. Initialize $Viterbi_0(y_0) = 1$.
  2. For $t = 1, \ldots, T$, compute $Viterbi_t(y_t) = max_{y_{t-1}} Viterbi_{t-1}(y_{t-1}) G_t(y_{t-1}, y_t; \mathbf{x}, \theta)$
  3. Return $max_{y_T} Viterbi_T(y_T)$.

  To recover the actual sequence $\mathbf{y}*$, we work backwards from the value of y_T that maximizes $Viterbi_T$, back to the optimal assignment of y_1.

  1. Compute $y_T^* = argmax_{y_T} Viterbi_T(y_T)$
  2. For $t = T, \ldots, 2$, compute $y_{t-1}^* = argmax_{y_{t-1}} Viterbi_{t-1}(y_{t-1}) G_t(y_{t-1}, y_t^*; \mathbf{x}, \theta)$

  Implement the computation of $Viterbi_t$ and the recovery of the best sequence in the function `computeViterbi` in `submission.py`. Once you have implemented `computeViterbi`, you can run the following command to get an interactive shell to play around with your CRF.

```
>> python run.py shell --parameters data/english.binary.crf --featureFunction binary
>> viterbi Mark had worked at Reuters
          -PER- -O- -O- -O- -ORG
```

- (10 points) Next, let us compute forward and backward messages. Here is the algorithm for computing forward messages. To prevent numerical underflow/overflow errors, we normalize $Forward_t$ and keep track of the log normalization constant A.

  1. Initialize $Forward_0(y_0) = 1$ and $A = 0$.
  2. For $t = 1, \ldots, T$ do

- Compute $Forward_t(y_t) = \sum_{y_{t-1}} Forward_{t-1}(y_{t-1}) G_t(y_{t-1}, y_t; \mathbf{x}, \theta)$
- Update $A \leftarrow A + log(\sum_{y_t} Forward_t(y_t))$
- Normalize: $Forward_t(y_t) \leftarrow \frac{Forward_t(y_t)}{\sum_{y'_t} Forward_t(y'_t)}$

3. Return A, which equals the log of the normalization constant $Z(\mathbf{x}; \theta)$.

Implement `computeForward`, which returns the log normalization constant A as well as the normalized forward messages $[Forward_1(y), \ldots, Forward_T(y)]$. We have provided `computeBackward` which produces a sequence of normalized backward messages $[Backward_1(y), \ldots, Backward_T(y)]$.

- (10 points) Given the forward and backward messages, we can combine them to compute marginal probabilities:

$$P(y_{t-1}, y_t | \mathbf{x}, \theta) = \frac{Forward_{t-1}(y_{t-1}) G_t(y_{t-1}, y_t; \mathbf{x}, \theta) Backward_t(y_t)}{Z(\mathbf{x}; \theta)}$$

Implement `computeEdgeMarginals` that will compute $P(y_{t-1}, y_t | \mathbf{x}, \theta)$. You should use `computeForward` and `computeBackward`.

We have implemented the learning algorithm that uses these marginals to compute a gradient. You can train the CRF (with standard features explained in the next subsection) now by running:

```
>> python run.py train --featureFunction binary --output-path my.crf
```

It could take up to 10-20 minutes to train, so only do this after you are confident that your code works. The program will write the parameters of the trained CRF to `my.crf`. You should get a dev F1 score of around 56.7%, which is quite poor. In the next section, we will design better features that will substantially improve the accuracy.

Problem 2.2: Named-entity recognition (10 points)

In the previous part, we developed all the algorithms required to train and use a linear-chain CRF. Now we turn to designing better features. We are using a subset of the CoNLL 2003 dataset consisting of 2,000 sentences with the following NER tags: `-PER-` (person), `-LOC-` (location), `-ORG-` (organization), `-MISC-` (miscellaneous), and `-O-`(other). We have provided two very simple feature functions. We will describe the feature functions using the example three-word sentence: `xs = ["Beautiful", "2", "bedroom"]`.

- `unaryFeatureFunction(t, y_, y, xs)` introduces an indicator feature for the current tag `y` and the current word `xs[t]`. For example, `unaryFeatureFunction(2, "-FEAT-", "-SIZE-", xs)` would return { `("-SIZE-", "bedroom") : 1.0` }. Note that `"-SIZE-"` is `y` and `x[2]` is `"bedroom"`.
- `binaryFeatureFunction(t, y_, y, xs)` includes all the features from `unaryFeatureFunction` and introduces another indicator feature for the previous tag `y_` and the current tag `y`. For example, `binaryFeatureFunction(2, "-FEAT-", "-SIZE-", xs)` would return, { `("-FEAT-", "-SIZE-") : 1.0, ("-SIZE-", "bedroom") : 1.0` }. Note that `"-FEAT-"` is `y_` and `"-SIZE-"` is `y`.

To train a model, use the following command,

```
>> python run.py train --featureFunction [unary|binary] --output-path my.crf
```

Use one of the featureFunctions `unary` or `binary` to do the training. As the model trains, it will print the likelihood of the training data at each iteration as well as a confusion matrix and F1 score for the NER tags. If you specify an output path, you can interact with the CRF you trained by providing the path as an argument to the shell,

```
>> python run.py shell --parameters my.crf --featureFunction [unary|binary]
```

Remember to use the same feature function as the one you used to train!

A common problem in NLP is making accurate predictions for words that we've never seen during training (e.g., Punahou). The reason why our accuracy is so low is that all our features thus far are defined on entire words, whereas we would really like to generalize. Fortunately, a CRF allows us to us to choose arbitrary feature functions $\phi_{local}(t, y_{t-1}, y_t, \mathbf{x})$. Next, we will define features based on capitalization or suffixes, which allow us to generalize to unseen words, as well as features that look at the current tag and the previous and next words, to capture more context.

Implement `nerFeatureFunction(t, y_, y, xs)` with the features below. Again, we will illustrate the expected output using the sentence: `xs = ["Beautiful", "2", "bedroom"]`. For convenience, think of the sentence as being padded with special begin/end words: `xs` is `-BEGIN-` at position -1 and `-END` at position `len(xs)`.

- All the features from `binaryFeatureFunction`.
- An indicator feature on the current tag `y` and the capitalization of the current word `xs[t]`. For example, `nerFeatureFunction(0, "-BEGIN-", "-FEAT-", xs)` would include the following features, {`("-FEAT-", "-CAPITALIZED-"):1.0`}. On the other hand, `nerFeatureFunction(2, "-SIZE-", "-SIZE-", xs)` would not add any features because `"bedroom"` is not capitalized.
- An indicator feature on the current tag `y` and the previous word `xs[t-1]`. For example, `nerFeatureFunction(2, "-SIZE-", "-SIZE-", xs)` would add: {`("-SIZE-", "PREV:2"):1.0`}. This is because `2` occurs before `"bedroom"` which is `xs[2]`. And `nerFeatureFunction(0, "-BEGIN-", "-FEAT-", xs)` would add {`("-FEAT-", "PREV:-BEGIN-"):1.0`}.
- A similar indicator feature for the current tag `y` and next word `xs[t+1]`; For example, `nerFeatureFunction(0, "-BEGIN-", "-FEAT-", xs)` would include, {`("-FEAT-", "NEXT:2"):1.0`}, and `nerFeatureFunction(2, "-SIZE-", "-SIZE-", xs)` would include, {`("-SIZE-", "NEXT:-END-"):1.0`}.
- Repeat the above two features except using capitalization instead of the actual word; consider `-BEGIN-` and `-END-` to be uncapitalized. An an example, `nerFeatureFunction(1, "-SIZE-", "-SIZE-", ["Beautiful", "2", "Bedroom"])` would include {`("-SIZE-", "-PRE-CAPITALIZED-"):1.0, ("-SIZE-", "-POST-CAPITALIZED-"):1.0`}.

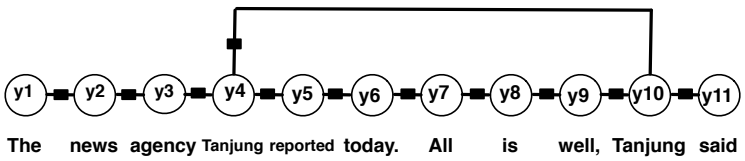
After you verify that your features are working using grader.py, train your CRF:

```
>> python run.py train --featureFunction ner --output-path ner.crf
```

Your dev F1 score should be around 71%.

Problem 2.3: Handling long-range dependencies (20 points) EC for comp440/required for comp557

Consider the following example,



It is clear that in the first occurrence of the word, "Tanjung" is an organization. Given only the second sentence though, it is ambiguous whether "Tanjung" is a person or organization. In fact, the CRF we previously trained predicts "Tanjung" to be a person here. To address this, we would like to add a constraint that all occurrences of a word are tagged the same. The catch is that such a constraint introduces long-range dependencies between the tag variables, complicating inference.

Gibbs sampling for linear chain CRFs (5 points)

Before we look at a Gibbs sampler to handle long-range dependencies, let us revisit the linear-chain CRF that we studied in the first part of this problem. Recall that Gibbs sampling updates y_t by sampling from its conditional distribution, given the values of the rest of the variables, $y_{-t} = (y_1, \ldots, _{t-1}, y_{t+1}, \ldots, y_T)$. Write an expression for the conditional distribution $P(y_t|y_{-t}, \mathbf{x}; \theta)$ for the linear-chain CRF in terms of the potentials G_t. Place your expression in `writeup.pdf`.

Implementing Gibbs sampling for linear chain CRFs (15 points)

We have provided you a function `gibbsRun()` in `submission.py` which provides the framework for the Gibbs sampler. Read the documentation for this function first. Now implement the following.

- (5 points) Implement `chooseGibbsCRF` that samples a value for y_t based on its conditional distribution you derived above, and reassigns y_t to that value. Note that you should only use the potential between y_t and its Markov blanket.
- (5 points) Implement `computeGibbsProbabilities` that estimates the probability for each output sequence based on the samples of the Gibbs sampler.
- (5 points) Implement `computeGibbsBestSequence` that estimates the most likely sequence (the interface is similar to `computeViterbi`).

Once you have implemented these functions, you can run the following command(s) to look at the output of Gibbs.

```
$ python run.py shell --parameters data/english.binary.crf --featureFunction binary
>> gibbs_best Mark had worked at Reuters
```

```
-PER--O--O--O--ORG-
>> gibbs_dist Mark had worked at Reuters
0.622  -PER--O--O--O--ORG-
0.1902  -PER--O--PER--O--ORG-
0.124  -PER--O--ORG--O--ORG-
0.0321  -ORG--O--O--O--ORG-
0.0084  -ORG--O--PER--O--ORG-
```

Your numbers will not match exactly due to the randomness in sampling.

3 Text classification (40 points)

According to Microsoft, circa 2007, 97% of all email is unwanted spam. Fortunately, most of us avoid the majority of these emails because of well-engineered spam filters. In this assignment, we will build a text classification system that, despite its simplicity, can identify spurious email with impressive accuracy. You will then apply the same techniques to identify positive and negative product reviews and to classify email posts into topical categories. All the code for this problem resides in folder `text_classification`

Problem 3.1: Spam classification (20 points)

Let us start by building a simple rule-based system to classify email as spam or ham. To test our system, we will be using a corpus of email made publicly available after the legal proceedings of the Enron collapse; within the `data/spam-classification/train` directory, you will find two folders `spam` and `ham`. Each folder contains a number of full text files that contain the whole email without headers and have been classified as spam and ham respectively.

In order to implement your own spam classifier, you will subclass `Classifier` and implement the `classify` method appropriately in `submission.py`. As usual, `grader.py` contains a number of simple test cases for your code. To run, type `python grader.py` on the command line.

Additionally, we have provided a script `main.py` to help you interactively run your code with different parameters; be ready to change this script to use alternate features, print different statistics, etc. It is really just meant to help you get started. To use this script, type `python main.py part<part>`, using the section number (2.1, 2.2, etc.). `python main.py part<part> -h` lists additional parameters you might find useful. The script will output the classification error rate as well as a confusion matrix for the training and development set. Each cell of the confusion matrix, (row, column), of the matrix contains the number of elements with the true label row that have been classified as column.

Problem 3.1.1: Rule-based system (3 points)

- (2 points) `data/spam-classification/blacklist.txt` contains a list of words sorted by descending spam correlation. Implement `RuleBasedClassifier` by discarding any email containing at least one word from this list. It is recommended you store the blacklist as a

set to reduce lookup time. For comparison, our system achieved a dev error rate of about 48% on the training set with this heuristic. You can test this by running `python main.py part3.1.1` at the command line. If you run `python grader.py` you should get train error of 0.474 and a dev error of .48

- (1 point) Relax this heuristic by only discarding email that contains at least n or more of the first k listed words. You can test this by running `python main.py part3.1.1 -n 1 -k 10000` at the command line for $n = 1$ and $k = 10000$. Report your dev error results on the training set in a 3x3 table with $n = 1, 2, 3$ and $k = 10000, 20000, 30000$.

Problem 3.1.2: Linear classifiers (4 points)

As you have observed, this naive rule-based system is quite ineffective. A reasonable diagnosis is that each word, whether it is viagra or sale, contributes equally to the 'spamminess' of the email. Let us relax this assumption by weighing each word separately.

Consider producing a spamminess score, $f_w(x)$ for each email document x which sums the 'spamminess' scores for each word in that document;

$$f_w(x) = \sum_{i=1}^{L} w(word_i)$$

where L is the length of the document x, and w is the 'spamminess' score for word $word_i$. We can use a linear classifier that will classify the email as spam if $f_w(x) \geq 0$, and as ham, otherwise.

Note that the order of words does not matter when computing the sum $f_w(x)$. Thus, it is convenient to represent a document as a sparse collection of words. An ideal data structure for this is a hash map or dictionary. For example, the document text `The quick dog chased the lazy fox over the brown fence`, would be represented using the dictionary, { `brown`: 1, `lazy`: 1, `fence`: 1, `fox`: 1, `over`: 1, `chased`: 1, `dog`: 1, `quick`: 1, `the`: 2, `The`: 1}.

Let the above vector representation of a document be $\phi(x)$; in this context, the vector representation is called a feature. The use of individual words or unigrams is a common choice in many language modeling tasks. With this vector or feature representation, the spamminess score for a document is simply the inner product of the weights and the document vector $f_w(x) = w.\phi(x)$.

Let the positive label be represented as a 1. Then, we can write the predicted output $\hat{y}$ of the classifier as

$$\hat{y} = \begin{cases} +1 & \text{if } w.\phi(x) \geq 0 \\ -1 & \text{if } w.\phi(x) < 0 \end{cases}$$

- (2 points) Implement a function `extractUnigramFeatures` that reads a document and returns the sparse vector $\phi(x)$ of unigram features. Run `python grader.py` to check your implementation.

- (2 points) Implement the `classify` function in `WeightedClassifier`. You can test whether you have implemented this correctly by running `python grader.py`.

Problem 3.1.3: Learning to distinguish spam (13 points)

The next question we need to address is where the vector of spamminess scores, w, comes from. Our prediction function is a simple linear function, so we will use the perceptron algorithm to learn weights. The perceptron algorithm visits each training example and incrementally adjusts weights to improve the classification of the current labelled example (x, y). If $\hat{y}$ is the prediction the algorithm makes with the current set of weights $w^{(t)}$, i.e., $\hat{y} = I(w^{(t)}.\phi(x) \geq 0)$, then if $\hat{y} \neq y$, increment $w^{(t)}$ by $y \times \phi(x)$.

- (6 points) Implement `learnWeightsFromPerceptron` that takes as input a corpus of training examples, and returns the w learned by the perceptron algorithm. Initialize your weights uniformly with 0.0. If you run `python grader.py` you should see a train error of 0.008968 and a dev error of 0.04868.
- (2 points) So far, we have looked at scores for single words or unigrams. We will now consider using two adjacent words, or bigrams as features by implementing `extractBigramFeatures`. To handle the edge case of a word at the beginning of a sentence (i.e. after a punctuation like '.', '!' or '?'), use the token `-BEGIN-`. On the previous example, `The quick dog chased the lazy fox over the brown fence`, `extractBigramFeatures` would return, {`the brown`: 1, `brown`: 1, `lazy`: 1, `fence`: 1, `brown fence`: 1, `fox`: 1, `over`: 1, `fox over`: 1, `chased`: 1, `dog`: 1, `lazy fox`: 1, `quick dog`: 1, `The quick`: 1, `the lazy`: 1, `chased the`: 1, `quick`: 1, `the`: 2, `over the`: 1, `-BEGIN- The`: 1, `dog chased`: 1}.

  Run `python grader.py` to check your implementation.
- (5 points) Vary the number of examples given to the training classifier in steps of 500 from 500 to 5,000. Provide a table of results showing the training and development set classification error when using bigram features. How did the additional features help the training and development set error? You can run `python main.py part3.1.3` to get the results for producing the table.

Problem 3.2: Sentiment classification (8 points)

You have just constructed a spam classifier that can identify spam with a 96% accuracy. While this in itself is great, what's really impressive is that the same system can easily be used to learn how to tackle other text classification tasks. Let us look at something that is completely different; identifying positive and negative movie reviews. We have provided a dataset in `data/sentiment/train`, with labels `pos` and `neg`.

- (3 points) Use the perceptron learning algorithm you wrote in the previous section to classify positive and negative reviews. Report the training and development set error rate for unigram features and bigram features. You can run `python main.py part3.2` to get these results. You can modify the function `part2` in main.py if you want to change parameters.
- (5 points) Make a table of the training and development set classification errors as you vary the number of iterations of the perceptron algorithm from 1 to 20. Use bigram features for this part. Use `main.py` for this part as before. Does development set error rate monotonically

decrease with iteration number? Why or why not? You might find it useful to write another version of `learnWeightsFromPerceptron` that prints training and development error in each iteration. Optionally, you might try plotting the errors to visually see how the two errors behave. We recommend the `matplotlib` library. Include these plots in your writeup if you make them.

Problem 3.3: Document categorization (12 points)

Finally, let's see how this system can be generalized to tasks with multiple labels. We will apply our knowledge to the task of categorizing emails based on topics. Our dataset in `data/topics/train` contains a number of emails from 20 popular USENET groups that have been segregated into 5 broader categories.

- (9 points) A common approach to multi-class classification is to train one binary classifier for each of the categories in a "one-vs-all" scheme. Namely, for the binary classifier i, all examples labelled i are positive examples, and all other examples are negative. When using this classifier for prediction, one uses the class label with the largest score, $f_i(x)$. Implement the `OneVsAllClassifier` classifier (and `MultiClassClassifier`). In order to train this classifier, you will also need to implement `learnOneVsAllClassifiers` that will appropriately train binary classifiers on the provided input data.
- (4 points) Report the train and development set error rate for unigram features and bigram features in your writeup. You can generate these by modifying function `part3` in `main.py`. If you run `python main.py part3.3` with our default `part3`, you should see a development error rate of about 12%.

4 Image classification (40 points)

In this assignment, you will implement an image classifier that distinguishes birds and airplanes. We will use the perceptron algorithm as a starting point but what should the features (arguably the most important part of machine learning) be? We will see that rather than specifying them by hand, as we did for spam filtering, we can actually learn the features automatically using the K-means clustering algorithm. We will be working with the CIFAR-10 dataset, one of the standard benchmarks for image classification. We assume that your version of Python has `numpy` and `PIL` packages already installed. If you do not have these packages, you can install them fairly easily from distributions downloadable from the Web. If you use Anaconda Python, these packages come pre-installed.

Warmup

Now we will get you familiar with the classification task. You will be classifying a 32×32 image as either a bird ($y = 1$) or a plane ($y = 0$). One way to do this is to train a classifier on the raw pixels, that is, $\phi(x) \in \Re^{32 \times 32}$ vector where each component of the vector represents the intensity of a particular pixel. Try running this classifier with

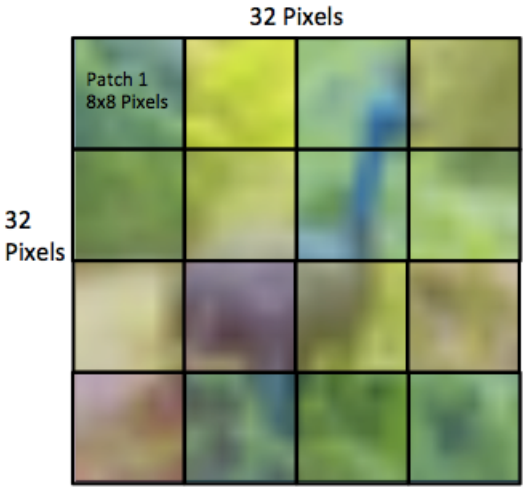


Figure 1: The sixteen patches of size 8x8 pixels corresponding to an image of size 32x32 pixels.

```
python run.py --pixels
```

As you can see, these features do not work very well, the classifier drastically overfits the training set and as a result barely generalizes better than chance.

The problem here is that pixel values themselves are not really meaningful. We need a higher-level representation of images. So we resort to the following strategy: Each image is a 32×32 grid of pixels. We will divide the image into sixteen 8×8 "patches". Now comes the key part: we will use K-means to cluster all the patches into centroids. These centroids will then allow us to use a better feature representation of the image which will make the classification task much easier.

Implementing the K-means algorithm (10 points)

In this part of the assignment you will implement K-means clustering to learn centroids for a given training set. Specifically you should fill out the method `runKMeans` in the file `submission.py`.

Let $D = \{x_1, \ldots, x_n\}$ be a set of data, where $x_i \in \Re^d$. We will construct K clusters with means $\mu_1, \ldots, \mu_K$ where $\mu_j \in \Re^d$.

- Initialize a set of means $\mu_1, \mu_2, \ldots, \mu_K$.
- for i in 1..maxIter
  - Assignment step: assign each x_i to the closest cluster mean $z_i \in \{\mu_1, \ldots, \mu_K\}$.

$$z_i = argmin_k ||x_i - \mu_k||_2$$

  - Update step: given the cluster assignments z_i, recalculate the cluster means μ's.

$$mu_k = \frac{1}{\sum_{z_i=k} 1} \sum_{z_i=k} x_i$$

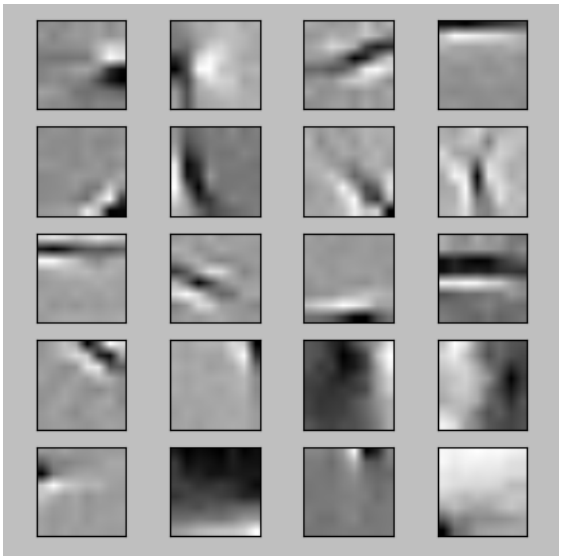


Figure 2: 20 centroids learned from K-means on patches from the first 1000 training images..

We start you off by initializing K-means with random centroids where each floating point number is chosen from a normal distribution with mean 0 and standard deviation 1. Test the K-means code with the provided test utility in `grader.py` by running:

```
python grader.py
```

Optional: One way to determine if your K-means algorithm is learning sensible features is to view the learned centroids using our provided utility function. To view the first 25 learned centroids, run

```
python run.py --view
```

Your centroids should look similar to Figure 2. Notice how the centroids look like edges. Important note on patches: Images are composed of patches which have been converted to gray-scale followed by standard image preprocessing. This includes normalizing them for luminance and contrast as well as further whitening.

Feature Extraction (10 points)

The next step in the pipeline is to use the centroids to generate features that will be more useful for the classification task then the raw pixel intensities. One way to do this is to represent each patch by the distance to the centroids that are closest to it, the intuition being that we can encode a patch by the centroids to which it is most similar. We will map each image x to its new feature vector $\phi(x) \in \Re^{16k}$, where there is a real value for each patch, centroid combination.

Let $p_{ij} \in \Re^{64}$ be the ij-th patch of x where $i, j = 1, \ldots, 4$. The relative activation, a_{ijk}, of centroid μ_k by patch p_{ij} is defined to be the average Euclidean distance from p_{ij} to all centroids minus the Euclidean distance from p_{ij} to μ_k. The Euclidean distance is the L_2 norm, $||v||_2 = \sqrt{v^T v}$.

$$a_{ijk} = \frac{1}{K}\left(\sum_{k'=1}^{K} ||p_{ij} - \mu_{k'}||_2\right) - ||p_{ij} - \mu_k||_2$$

The feature value for patch p_{ij} and centroid μ_k is the max of the relative activation and zero.

$$\phi_{ijk}(x) = max(a_{ijk}, 0)$$

Implement the function `extractFeatures` in `submission.py`. We will use these features in the linear classifier below.

Supervised Training (20 points)

The final task is to use our better feature representation to classify images as birds or planes. We have provided you with a working Perceptron classifier which does fairly well. You will implement the logistic and hinge loss updates to learn the parameters and see if either of these improves classification accuracy and which does better. First you can run the Perceptron classifier to see how it performs on the test set:

```
python run.py --gradient=perceptron
```

You should see a test accuracy result between 60%-65% - we can do better!

- (10 points) Implement the `logisticGradient` method in `submission.py`. You can test this method with `python grader.py`. Given example $(\phi(x), y)$ where $\phi(x) \in \Re^d$ and $y \in \{0, 1\}$, and parameter vector w, define $yy = 2 * y - 1$ to map $\{0, 1\}$ to $\{-1, 1\}$. The logistic loss function is

  $$Loss_{logistic}(w, \phi(x), yy) = log(1 + e^{-w^T \phi(x) * yy})$$

  Compute the gradient of this function with respect to w and complete the `logisticGradient` method.

- (10 points) Implement the `hingeLossGradient` method in `submission.py`. The hinge loss function is

  $$Loss_{hinge}(w, \phi(x), yy) = max(1 - w^T \phi(x) * yy, 0)$$

  Compute the gradient of this function with respect to w and complete the `hingeLossGradient` method. You can test this method with `python grader.py`.

Now you are ready to run the entire pipeline and evaluate performance. Run the full pipeline with:

```
python run.py --gradient=<loss function>
```

The loss function can be any of {"hinge","logistic","perceptron"}. The output of this should be the test accuracy on 1000 images. One benefit of using an unsupervised algorithm to learn features is that we can train the algorithm with unlabeled data which is much more easily obtained than labeled data. We have included an option to train the K-means algorithm using 500 extra unlabeled images. You can run this with

```
python run.py --gradient=hinge -m
```

The performance of the supervised classifier goes up even though we are not using labeled data – a major benefit to using an unsupervised algorithm to learn a feature representation!

Numpy cheat sheet

This is a quick overview of the functions you will need to complete this assignment. A really thorough introduction is at `http://wiki.scipy.org/Tentative_NumPy_Tutorial`.

All standard operations on vectors, matrices and n-dimensional arrays are element-wise in `numpy`. For example

```
A = numpy.random.randn(5,5)
B = numpy.random.randn(5,5)
A+B # element-wise addition
A*B # element-wise multiplication
```

You can also access individual elements, columns and rows of A using the following notation,

```
A[0,0] # first element of the first row of A
A[:,1] # second column of A
A[3:5,:] # fourth and fifth rows of A
```

In order to take the matrix product of A and B use the `numpy.dot` function.

```
numpy.dot(A,B) # matrix multiplication A*B
A.dot(B) # same as above
```

To find the minimum or maximum element in an array or along a specific axis of an array (rows or columns for 2D), use `numpy.minimum` or `numpy.maximum`

```
numpy.minimum(A) # min of A
numpy.minimum(A, axis=0) # min of each column of A
numpy.maximum(A, axis=1) # max of each row of A
```

To take the indices of the minimum element in each row or column of a 2D array use the `numpy.argmin` function and specify the axis

```
numpy.argmin(A,axis=0) # argmin of each column
```

Similarly you can take the mean along the columns or rows of a 2D array using `numpy.mean` and the sum along a specific axis using `numpy.sum`

```
numpy.mean(A,axis=0) # mean of each column of A
numpy.sum(A,axis=1) # sum of each row of A
```

5 Perceptrons, Decision Trees, Neural Networks (15 points) 15

The data set below is a subset of a census database (the full data is available at `ftp://ftp.ics.uci.edu/pub/machine-learning-databases/adult`). Your goal is to predict whether an individual earns more or less than 50K a year based on education, gender and citizenship. Education is a discrete-valued attribute which can take one of three values; BS, MS, PhD; gender takes one of two values: male, female; and citizenship has two values: US and nonUS. You will use four different algorithms to learn models over this training data set.

Education	Gender	Citizenship	Income
BS	male	US	$\leq$50K
MS	male	nonUS	>50K
BS	female	US	$\leq$50K
PhD	male	nonUS	$\leq$50K
MS	female	US	>50K
PhD	female	nonUS	$\leq$50K
BS	male	US	$\leq$50K
PhD	male	US	>50K
BS	female	nonUS	$\leq$50K
PhD	female	US	>50K

Test your models on the following test set.

Education	Gender	Citizenship
PhD	male	US
PhD	male	nonUS
MS	female	nonUS

a. The perceptron algorithm (5 points)

- (1 point) How would you encode the training data as inputs to a perceptron?
- (3 points) Initialize all weights to zero and perform one pass of perceptron training on the training set, doing updates in the order in which the observations are presented. Present your result in a table, showing the weight vector after processing each example.
- (1 point) Will the perceptron converge if you run it long enough? Justify your answer.

b. Decision Trees (5 points)

- (2 points) Calculate the information gain of each of the three features. Which feature should be chosen as the root of the decision tree?
- (2 points) Now continue with the decision tree construction process with the root attribute chosen above. Determine the attribute to split on for the next level of nodes in the decision tree. Show your information gain calculations. Continue the construction process till all the

leaf nodes have instances belonging to the same `Income` class. Show the final (unpruned) tree that you have constructed.

- (1 point) What does your decision tree predict on the three examples in the test set? That is, fill out the following table.

Education	Gender	Citizenship	Income
PhD	male	US	
PhD	male	nonUS	
MS	female	nonUS	

c. Neural networks (5 points)

- (1 point) How would you encode the data as inputs to a feedforward neural network?
- (3 points) Code this example with sklearn's MLP implementation, and train the network on the given training set. Documentation is available at `http://scikit-learn.org/stable/modules/neural_networks_supervised.html` You will need to set up two numpy arrays for training `trainX` and `trainy` corresponding to the 10 training examples. You will also need to set up the test set as the array `testX`. You will have to make a choice of the number of hidden units. Try the range $2, \ldots, 5$ and report the cross-validated training error.
- (1 point) What are the predictions made on the test set? Compare the classifications of the test cases made by the four classifiers and comment on their similarities and differences.