Assignment 1

Due: Mon 04 Dec 2017 Midnight

Natural Language Processing - Fall 2018 Michael Elhadad

This assignment covers 2 parts:

1. Language models.
2. Linear regression in a probabilistic model and regularization

The objectives of the assignment are for Part 1:

1. Learn how to use Python
2. Learn how to access a text corpus using NLTK corpus classes
3. Learn how to use n-gram models to generate a language model
4. Learn how to compare different language models using perplexity
5. Learn how to compare different smoothing methods for n-gram distribution estimation
6. Compare word-based and character-based language models
7. Compare the expressive power of n-gram models and more complex neural-network language models.

For Part 2, on distributions, regression and classification, the objectives are:

1. Learn about basic statistical distributions (Normal, Multinomial), expectations, sampling and estimation.
2. Learn how to perform regression using a synthetic dataset.
3. Develop an intuition of how regularization helps overcome overfitting.

Make sure you have installed scipy, scikit-learn and numpy to work on this assignment. This should be already available if you have installed the Anaconda distribution.

Submit your solution in the form of an iPython (Jupyter) notebook file (with extension ipynb). Images should be submitted as PNG or JPG files. The whole code should also be submitted as a separate folder with all necessary code to run the questions separated in clearly documented functions.

Look at the following two notebooks as examples to get you started - copy them to your Anaconda folder, then start your iPython notebook server and explore them:

- iPython Notebook showing a plot (ipynb)
- iPython Notebook skeleton (ipynb)

Content

Part 1: Language Models

In this question, we will develop a language model over two distinct datasets. One language models will predict words given a history of previous words in the text. The other one will predict characters given a history of previous characters. We will use n-gram models, implemented using two distinct methods, and evaluate the performance of the models using perplexity.

1.1 Data Exploration

1.1.1 Gathering and Cleaning Up Data

We will build a language model of English based on a dataset which contains a collection of sentences. Usually, documents are collected from natural sources and include formatting characters which are not related to the text itself (for example, html tags). Before we can use the text, we must clean it up. In this question, we want to get a feeling of how difficult it is to clean real-world data. Please read the tutorial in Chapter 3 of the NLTK book. This chapter explains how to access "raw text" and clean it up: remove HTML tags, segment in sentences and in words.

Lookup at the data of the Brown corpus as it is stored in the nltk_data folder (by default, it is in a folder named like C:\nltk_data\corpora\brown under Windows). The format of the corpus is quite simple. We will attempt to add a new "section" to this corpus.

A dataset used in many articles studying language models is the Penn Tree Bank (PTB) dataset, which contains 929k training words, 73k validation words, and 82k test words. It has the 10k most frequent words in its vocabulary. Tomas Mikolov's webpage distributes a version of this dataset which has been pre-processed such that only the top-10K most frequent words occur, the others are replaced by the token <unk>, words are separated with spaces according to consistent tokenization rules, numbers are replaced by a single token N, and sentences are segmented one per line.

For example:

```
consumers may want to move their telephones a little closer to the tv set
<unk> <unk> watching abc 's monday night football can now vote during <unk> for the greatest play in N years
from among four or five <unk> <unk>
```

The dataset is also available in a form that is easy to process when building character-level language models (where the objective is to predict the next char, not the next word, given a history of the previous characters):

```
c o n s u m e r s _ m a y _ w a n t _ t o _ m o v e _ t h e i r _ t e l e p h o n e s _ a _ l i t t l e _ c l o s e r _
t o _ t h e _ t v _ s e t
< u n k > _ < u n k > _ w a t c h i n g _ a b c _ ' s _ m o n d a y _ n i g h t _ f o o t b a l l _ c a n _ n o w _
v o t e _ d u r i n g _ < u n k > _ f o r _ t h e _ g r e a t e s t _ p l a y _ i n _ N _ y e a r s _ f r o m _
a m o n g _ f o u r _ o r _ f i v e _ < u n k > _ < u n k >
```

These types of low-level text encoding are critical to any process of NLP.

Your task is to take as input an arbitrary text file and to format it in a way that applies the same conventions as the word-level Penn Treebank representation of Mikolov:

- Keep only the top-10K most frequent words in the dataset
- Replace all numbers by the token N
- Tokenize the words: observe how "don't" and "caller's" and "$12" are split into two words.
- Remove all punctuations
- Segment the sentences and print them one by line
- Output all the text in lowercase

You can use the nltk utilities for tokenization and sentence segmentation documented here.

To identify the top-10K most frequent words, you can use the Python collections.Counter() object and its method most_common(n). You must submit:

1. The code of the method ptb_preprocess(filenames, top=10000) which given a list of filenames pre-processes the files and outputs the re-formatted files with an additional file extension (for example, file1.txt as input will generate file1.txt.out).
2. Example files that demonstrate each of the normalization transformations requested.

1.1.2 Gathering Basic Statistics

Before we apply any form of machine learning methods on an input dataset, we must explore the dataset and gather descriptive statistics about it. We want to collect and plot the following information on a dataset:

- The total number of tokens
- The total number of characters
- The total number of distinct words (vocabulary)
- The total number of tokens corresponding to the top-N most frequent words in the vocabulary
- The token/type ratio in the dataset
- The number of types that appear in the dev data but not the training data (hint: use sets for this)
- The average number and standard deviation of characters per token
- The total number of distinct n-grams (of words) that appear in the dataset for n=2,3,4.
- The total number of distint n-grams of characters that appear for n=2,3,4,5,6,7.

Word count distributions are said to follow power law distributions. In practice, this means that a plot of the log-frequency against the log-rank is nearly linear. Verify that this holds for the Penn Treebank dataset by constructing the appropriate corpus_counts counter:

```
# you need matplotlib version 1.4 or above
import matplotlib.pyplot as plt
import matplotlib
print(matplotlib.__version__)
%matplotlib inline

plt.loglog([val for word,val in corpus_counts.most_common(4000)])
plt.xlabel('rank')
plt.ylabel('frequency');
```

1.2 n-gram Language Model

Write Python functions to construct a word n-gram model given a dataset. The requested function is:

1. train_word_lm(dataset, n=2)

Describe the data structure you use for the model. How much memory do you expect a model to occupy? (Refer to the statistics results above and provide worst-case estimates as well as expected).

You can base your implementation either on the model described in this article (which needs to be adapted from characters to words) or in the nltk module nltk.model.ngram.

1.3 Language Model Evaluation

We now evaluate the performance of the learned language models by using two techniques: measuring perplexity on a test dataset and using the model to generate random text, then assessing the readability of the generated text.

1.3.1 Perplexity

Implement a Python function to measure the perplexity of a trained model on a test dataset. Adapt the methods to compute the cross-entropy and perplexity of a model from nltk.model.ngram to your implementation and measure the reported perplexity values on the Penn Treebank validation dataset.

One way to improve the model is to use a smoothing technique to increase the generalization of the trained model. Learn how the nltk probability distribution module provides different estimators that implement different

smoothing methods (Laplace, Lidstone, Witten-Bell, Good-Turing). Change your model to use a different estimator than the Maximum Likelihood Estimator (MLE) count-based estimator to compute the probability of p(w|history). Compare the obtained perplexity of the trained model on the development dataset for different estimators Lidstone for a variety of hyper-parameter gamma (0 < gamma < 1). Draw a graph of the obtained perplexity (or cross-entropy) on the dev test for different values of gamma.

Another way to improve the model is to use an n-gram model with increasing values of n (2,3,...10). Draw a graph of the obtained perplexety (or cross-entropy) on the dev test for different values of n between 2 and 20.

Based on the 2 graphs above, prepare the best predicted n-gram model based on a Lidstone model with an optimized gamma parameter and of the best possible n order. Test this model on the test dataset of the Penn Treebank and report results. Compare your result with the expected results on a uniform distribution of words (worst case) and on recent research results reported in research paper on language models that you can find in Google Scholar tested on the Penn Treebank dataset.

1.3.2 Generating Text from a Language Model

Another way to evaluate a language model is to use the model in a generative manner - that is, to randomly sample sentences starting from a seed prefix, and generating each next word by sampling from the model distribution p(w | prefix).

Implement the method generate(model, seed) and test it on the best model you have trained. Discuss ways to decide when the generation should stop. Report at least 5 randomly generated segments and comment on what you observe.

1.4 RNN language model

It is interesting to compare word-based and character-based language models. On the one hand, character-based models need to predict a much smaller range of options (one character out of ~100 possible characters vs. one word out of 200K possible words). On the other hand, we need to maintain a much longer history of characters to obtain a significant memory of the context which would make sense semantically.

Read the following article: The Unreasonable Effectiveness of Recurrent Neural Networks, May 21, 2015, Andrej Karpathy (up to Section "Further Reading"). Write a summary of this essay of about 200 words.

Read the follow-up article: The unreasonable effectiveness of Character-level Language Models (and why RNNs are still cool), Sept 2015, Yoav Goldberg. Write a summary of this essay of about 200 words.

Strikingly realistic output can be generated when training a character language-model on a strongly-constrained genre of text like cooking recipes. Train Yoav Goldberg's n-gram model on the dataset provided in do androids dream of cooking? which contains about 32K recipes gathered from the Internet.

Your task is:

1. Submit the 2 summaries of the articles above.
2. Gather the recipes dataset and prepare a dataset reader according to the structure of the files.
3. Report basic statistics about the dataset (number of recipes, tokens, characters, vocabulary size, distribution of the size of recipes in words and in chars, distribution of length of words).
4. Split the dataset into training, dev and test as a 80%/10%/10% split. Provide a Python interface to access the split conveniently.
5. Choose the order of the char n-gram according to the indications given in Yoav Goldberg's article. Justify the choice (you should use the dev test for this).
6. Train a char language model using either Yoav Goldberg's code (n-gram) or the PyTorch port of Andrej Karpathy's code in Char-rnn-pytorch (the neural network model will take several hours of runtime to converge on a CPU). You must install "pip install unidecode" in order to use this project.
7. Report on the perplexity of the trained language model.
8. Sample about 5 generated recipes from the trained language model.
9. Write 3 to 5 observations about the generated samples.

Part 2: Polynomial Curve Fitting

As a pre-requisite to this question, read this introduction to probability.

In this question, we will reproduce the polynomial curve fitting example used in Bishop's book in Chapter 1. Polynomial curve fitting is an example of a learning task called **regression** (as opposed to **classification** we discussed in class). In regression, the objective is to learn a function mapping an input variable X to a continuous target variable Y, while in classification, Y is a discrete variable. For educational purposes, it turns out that regression is simpler to grasp than classification. This task shows how to develop a probabilistic model of a task often described in geometric terms.

The outline of the task is:

1. Generate a synthetic dataset of N points (x, t) for a known function y(x) with some level of noise.
2. Solve the curve fitting regression problem using error function optimization.
3. Observe the problems of over-fitting this method produces.
4. Introduce regularization to overcome over-fitting as a form of MAP estimation.
5. Finally, use Bayesian estimation to produce an interval estimation of the function y.

2.1 Synthetic Dataset Generation

Learn how to use the numpy.random package to sample random numbers from well-known distributions in this reference page. In particular, we will use in this question the Normal distribution: numpy.random.normal.

Generate a dataset of points in the form of 2 vectors x and t of size N where:

```
ti = y(xi) + Normal(mu, sigma)

where the xi values are equi-distant on the [0,1] segment (that is, x_1 = 0, x_2=1/N-1, x_3=2/N-1..., x_N = 1.0)
mu = 0.0
sigma = 0.03
y(x) = sin(2πx)
```

Our objective will be to "learn" the function y from the noisy sparse dataset we generate.

The function **generateDataset(N, f, sigma)** should return a tuple with the 2 vectors x and t.

Draw the plot (scatterplot) of (x,t) using matplotlib for N=100. Look at the documentation of the numpy.random.normal function in Numpy for an example of usage. Look at the definition of the function numpy.linspace to generate your dataset.

Note: a useful property of Numpy arrays is that you can apply a function to a Numpy array as follows:

```
import math
import numpy as np
def s(x): return x**2
def f(x): return math.sin(2 * math.pi * x)
vf = np.vectorize(f)        # Create a vectorized version of f

z = np.array([1,2,3,4])

sz = s(z)                   # You can apply simple functions to an array
sz.shape                    # Same dimension as z (4)

fz = vf(z)                  # For more complex ones, you must use the vectorized version of f
fz.shape
```

2.2 Polynomial Curve Fitting

We will attempt to learn the function y given a synthetic dataset (x, t). We assume that y is a polynomial of degree M - that is:

$$y(x) = w_0 + w_1x + w_2x^2 + ... + w_Mx^M$$

Our objective is to estimate the vector $w = (w_0...w_M)$ from the dataset (x, t).

We first attempt to solve this regression task by optimizing the square error function (this method is called **least squares**:

$$\text{Define: } E(w) = 1/2\sum_i(y(x_i) - t_i)^2$$
$$= 1/2\sum_i(\sum_k w_k x_i^k - t_i)^2$$

If $t = (t_1, ..., t_N)$, then define the **design matrix** to be the matrix Φ such that $\Phi_{nm} = x_n^m = \Phi_m(x_n)$. We want to minimize the error function, and, therefore, look for a solution to the linear system of equations:

$$dE/dw_k = 0 \text{ for } k = 0..M$$

When we work out the partial derivations, we find that the solution to the following system gives us the optimal value w_{LS} given (x, t):

$$w_{LS} = (\Phi^T\Phi)^{-1}\Phi^T t$$

(Note: Φ is a matrix of dimension Nx(M+1), w is a vector of dimension (M+1) and t is a vector of dimension N.)

Here is how you write this type of matrix operations in Python using the Numpy library:

```
import numpy as np
import scipy.linalg

t = np.array([1,2,3,4])                    # This is a vector of dim 4
t.shape                                    # (4,)
phi = np.array([[1,1],[2,4],[3,3],[2,4]])  # This is a 4x2 matrix
phi.shape                                  # (4, 2)
prod = np.dot(phi.T, phi)                  # prod is a 2x2 matrix
prod.shape                                 # (2, 2)
i = np.linalg.inv(prod)                    # i is a 2x2 matrix
i.shape                                    # (2, 2)
m = np.dot(i, phi.T)                       # m is a 2x4 matrix
m.shape                                    # (2, 4)
w = np.dot(m, t)                           # w is a vector of dim 2
w.shape                                    # (2,)
```

Implement a method **OptimizeLS(x, t, M)** which given the dataset (x, t) returns the optimal polynomial of degree M that approximates the dataset according to the least squares objective. Plot the learned polynomial $w^*_M(x_i)$ and the real function sin(2πx) for a dataset of size N=10 and M=1,3,5,10.

2.3 Polynomial Curve Fitting with Regularization

We observe that the solution to the least-squares optimization has a tendency to over-fit the dataset. To avoid over-fitting, we will use a method called **regularization**: the objective function we want to optimize will take into account the least-squares error as above, and in addition the complexity of the learned model w.

We define a new objective function:

Define $E_{PLS}(w) = E(w) + \lambda E_W(w)$

Where E_{PLS} is called the penalized least-squares function of w

and E_W is the penalty function.

We will use a standard penalty function:

$$E_W(w) = 1/2\ w^T.w = 1/2 \sum_{m=0..M} w_m^2$$

When we work out the partial derivatives of the minimization problem, we find in closed form, that the solution to the penalized least-squares is:

$$\mathbf{w_{PLS}} = (\Phi^T\Phi + \lambda \mathbf{I})^{-1}\Phi^T t$$

λ is called a hyper-parameter (that is, a parameter which influences the value of the model's parameters w). Its role is to balance the influence of how well the function fits the dataset (as in the least-squares model) and how smooth it is.

Write a function **optimizePLS(x, t, M, lambda)** which returns the optimal parameters w_{PLS} given M and lambda.

We want to optimize the value of λ. The way to optimize is to use a development set in addition to our training set.

To construct a development set, we will extend our synthetic dataset construction function to return 3 samples: one for training, one for development and one for testing. Write a function **generateDataset3(N, f, sigma)** which returns 3 pairs of vectors of size N each, (x_{test}, t_{test}), ($x_{validate}$, $t_{validate}$) and (x_{train}, t_{train}). The target values are generated as above with Gaussian noise N(0, sigma).

Look at the documentation of the function numpy.random.shuffle() as a way to generate 3 subsets of size N from the list of points generated by linspace.

Given the synthetic dataset, optimize for the value of λ by varying the value of log(λ) from -40 to -20 on the development set. Draw the plot of the normalized error of the model for the training, development and test for the case of N = 10 and the case of N=100. The normalized error of the model is defined as:

$$NE_{\mathbf{w}}(\mathbf{x}, \mathbf{t}) = 1/N\ [\sum_{i=1..N}[t_i - \sum_{m=0..M} w_m x_i^m]^2]^{1/2}$$

Write the function **optimizePLS(xt, tt, xv, tv, M)** which selects the best value λ given a dataset for training (xt, tt) and a development test (xv, tv). Describe your conclusion from this plot.

2.4 Probabilistic Regression Framework

We now consider the same problem of regression (learning a function from a dataset) formulated in a probabilistic framework. Consider:

$$t_n = y(x_n; \mathbf{w}) + \varepsilon_n$$

We now model the distribution of ε_n as a probabilistic model:

$$\varepsilon_n \sim N(0, \sigma^2)$$

since $t_n = y(x_n; \mathbf{w}) + \varepsilon_n$:

$$p(t_n \mid x_n, \mathbf{w}, \sigma^2) = N(y(x_n; \mathbf{w}), \sigma^2)$$

We now assume that the observed data points in the dataset are all drawn in an independent manner (iid), we can then express the likelihood of the whole dataset:

$$p(\mathbf{t} \mid \mathbf{x}, \mathbf{w}, \sigma^2) = \prod_{n=1..N} p(t_n \mid x_n, \mathbf{w}, \sigma^2)$$
$$= \prod_{n=1..N} (2\pi\sigma^2)^{-1/2} \exp[-\{t_n - y(x_n, \mathbf{w})\}^2 / 2\sigma^2]$$

We consider this likelihood as a function of the parameters (**w** and σ) given a dataset (**t**, **x**).

If we consider the log-likelihood (which is easier to optimize because we have to derive a sum instead of a product), we get:

$$-\log p(\mathbf{t} \mid \mathbf{w}, \sigma^2) = N/2 \log(2\pi\sigma^2) + 1/2\sigma^2 \sum_{n=1..N} \{t_n - y(x_n; \mathbf{w})\}^2$$

We see that optimizing the log-likelihood of the dataset is equivalent to minimizing the error function of the least-squares method. That is to say, the least-squares method is understood as the maximum likelihood estimator (MLE) of the probabilistic model we just developed, which produces the values $\mathbf{w_{ML}} = \mathbf{w_{LS}}$.

We can also optimize this model with respect to the second parameter σ^2 which, when we work out the derivation and the solution of the equation, yields:

$$\sigma^2_{ML} = 1/N \sum_{n=1..N} \{y(x_n, \mathbf{w_{ML}}) - t_n\}^2$$

Given $\mathbf{w_{ML}}$ and σ^2_{ML}, we can now compute the **plugin posterior predictive distribution**, which gives us the probability distribution of the values of t given an input variable x:

$$p(t \mid x, \mathbf{w_{ML}}, \sigma^2_{ML}) = N(t \mid y(x, \mathbf{w_{ML}}), \sigma^2_{ML})$$

This is a richer model than the least-squares model studied above, because it not only estimates the most-likely value t given x, but also the precision of this prediction given the dataset. This precision can be used to construct a confidence interval around t.

We further extend the probabilistic model by considering a Bayesian approach to the estimation of this probabilistic model instead of the maximum likelihood estimator (which is known to over-fit the dataset). We choose a prior over the possible values of **w** which we will, for convenience reasons, select to be of a normal form (this is a conjugate prior as explained in our review of basic probabilities):

$$p(\mathbf{w} \mid \alpha) = \prod_{m=0..M} (\alpha / 2\pi)^{1/2} \exp\{-\alpha/2 \, w_m^2\}$$
$$= N(\mathbf{w} \mid \mathbf{0}, 1/\alpha \mathbf{I})$$

This **prior distribution** expresses our degree of belief over the values that **w** can take. In this distribution, α plays the role of a hyper-parameter (similar to λ in the regularization model above).

The Bayesian approach consists of applying Bayes rule to the estimation task of the posterior distribution given the dataset:

$$p(\mathbf{w} \mid \mathbf{t}, \alpha, \sigma^2) = \text{likelihood} \cdot \text{prior} / \text{normalizing-factor}$$
$$= p(\mathbf{t} \mid \mathbf{w}, \sigma^2) p(\mathbf{w} \mid \alpha) / p(\mathbf{t} \mid \alpha, \sigma^2)$$

Since we wisely chose a conjugate prior for our distribution over **w**, we can compute the posterior analytically:

$$p(\mathbf{w} \mid \mathbf{x}, \mathbf{t}, \alpha, \sigma^2) = N(\mu, \Sigma)$$

where Φ is the design matrix as above:

$$\mu = (\Phi^T \Phi + \sigma^2 \alpha I)^{-1} \Phi^T t$$

$$\Sigma = \sigma^2(\Phi^T\Phi + \sigma^2\alpha I)^{-1}$$

Given this approach, instead of learning a single point estimate of **w** as in the least-squares and penalized least-squares methods above, we have inferred a distribution over all possible values of **w** given the dataset. In other words, we have updated our belief about **w** from the prior (which does not include any information about the dataset) using new information derived from the observed dataset.

We can determine **w** by maximizing the posterior distribution over **w** given the dataset and the prior belief. This approach is called the **maximum posterior** (usually written MAP). If we solve the MAP given our selection of the normal conjugate prior, we obtain that the posterior reaches its maximum on the minimum of the following function of w:

$$1/2\sigma^2 \sum_{n=1..N}\{y(x_n, \mathbf{w}) - t_n\}^2 + \alpha/2\mathbf{w}^T\mathbf{w}$$

We find thus that $\mathbf{w_{MAP}}$ is in fact the same as the solution of the penalized least-squares method for $\lambda = \alpha\sigma^2$. In other words - this probabilistic model explains that the PLS is in fact the optimal solution to the problem when our prior belief on the parameters w is a Normal distribution N(0, 1/αI) which encourages small values for the parameter w.

A fully Bayesian approach, however, does not look for point-estimators of parameters like **w**. Instead, we are interested in the predictive distribution p(t | x, **x**, **t**). The Bayesian approach consists of marginalizing the predictive distribution over all possible values of the parameters:

$$p(t \mid x, \mathbf{x}, \mathbf{t}) = \int p(t|x, \mathbf{w})p(\mathbf{w} \mid \mathbf{x}, \mathbf{t})\, d\mathbf{w}$$

(For simplicity, we have hidden the dependency on the hyper-parameters α and σ in this formula.) On this simple case, with a simple normal distribution and normal prior over **w**, we can solve this integral analytically, and we obtain:

$$p(t \mid x, \mathbf{x}, \mathbf{t}) = N(t \mid m(x), s^2(x))$$

where the mean and variance are:

$$m(x) = 1/\sigma^2\ \mathbf{\Phi}(x)^T\mathbf{S} \sum_{n=1..N}\mathbf{\Phi}(x_n)t_n$$

$$s^2(x) = \sigma^2 + \mathbf{\Phi}(x)^T\mathbf{S}\mathbf{\Phi}(x)$$

$$\mathbf{S}^{-1} = \alpha\mathbf{I} + 1/\sigma^2 \sum_{n=1..N}\mathbf{\Phi}(x_n)\mathbf{\Phi}(x_n)^T$$

$$\mathbf{\Phi}(x) = (\Phi_0(x)\ \ldots\ \Phi_M(x))^T = (1\ x\ x^2\ \ldots\ x^M)^T$$

Note that the mean and the variance of this predictive distribution depend on x.

Your task: write a function **bayesianEstimator(x, t, M, alpha, sigma2)** which given the dataset (x, t) of size N, and the parameters M, alpha, and sigma2 (the variance), returns a tuple of 2 functions (m(x) var(x)) which are the mean and variance of the predictive distribution inferred from the dataset, based on the parameters and the normal prior over **w**. As you can see, in the Bayesian approach, we do not learn an optimal value for the parameter **w**, but instead, marginalize out this parameter and directly learn the predictive distribution.

Note that in Python, a function can return a function (like in Scheme) using the following syntax:

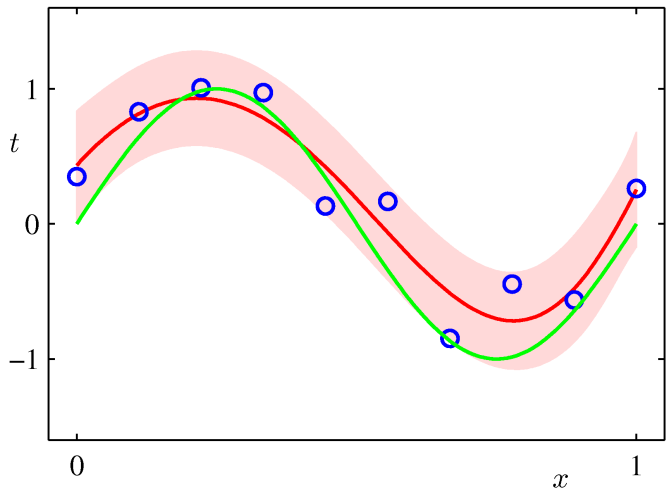
```
def adder(x):
    return lambda(y): x+y

a2 = adder(2)
```

```
print(a2(3))       // prints 5
print(adder(4)(3)) // prints 7
```

Draw the plot of the original function y = sin(2πx) over the range [0..1], the mean of the predictive distribution m(x) and the confidence interval (m(x) - var(x)$^{1/2}$) and (m(x) + var(x)$^{1/2}$) (that is, one standard deviation around each predicted point) for the values:

```
alpha = 0.005
sigma2 = 1/11.1
M = 9
```

over a synthetic dataset of size N=10 and N=100. The plot should look similar to the Figure below (from Bishop p.32).



Interpret the height of the band around the most likely function in terms of the distribution of the xs in your synthetic dataset. Can you think of ways to make this height very small in one segment of the function and large in another?

Last modified 19 Nov 2017