COMP3220 — Document Processing and the Semantic Web

Week 03 Lecture 1: Introduction to Text Classification

Diego Mollá

COMP3220 2020H1

Abstract

This lecture will focus on the task of text classification by using statistical classifiers. We will focus on the general workflow for applying statistical classifiers. In this lecture we will view statistical classifiers as black boxes.

Update March 4, 2020

Contents

Reading

- NLTK Book Chapter 6 "Learning to Classify Text"

Some Useful Extra Reading

- Ethem Alpaydin (2004). *Introduction to Machine Learning.* MIT Press.

1 What is Text Classification

Text Classification

What is Text Classification?
Classify documents into one of a *fixed predetermined* set of categories.

- The number of categories is predetermined.
- The actual categories is predetermined.

Examples

- Spam detection.
- Email filtering.
- Classification of text into genres.
- Classification of names by gender.
- Classification of questions.

Example: Spam Filtering

Distinguish this

```
Date:  Mon, 24 Mar 2008 From:  XXX YYY <xxx@yahoo.com> Subj:  Re:  Fwd:  MSc To:  Mark Dras <madras@ics.mq.edu.au>

   Hi, Thanks for that.  It would fit me very well to start 2009, its actually much better for me and I'm planning
to finish the project in one year (8 credit points).
```

from this

```
Date:  Mon, 24 Mar 2008 From:  XXX YYY <xxx@yahoo.co.in> Subj:  HELLO To:  madras@ics.mq.edu.au

   HELLO, MY NAME IS STEPHINE IN SEARCH OF A MAN WHO UNDERSTANDS THE MEANING OF LOVE AS TRUST AND FAITH IN EACH
OTHER RATHER THAN ONE WHO SEES LOVE AS THE ONLY WAY OF FUN ...
```

Classification Methods

Manual

- Web portals.
- Wikipedia.

Automatic

Hand coded rules

- e.g. 'Viagra' == SPAM.
- e.g. email filter rules.
- Fragile, breaks on new data.

Supervised learning

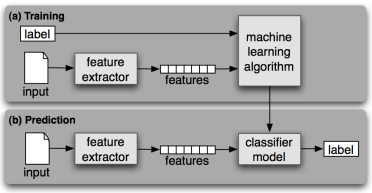
- Use an annotated corpus.
- Apply statistical methods.
- Greater flexibility.

Supervised Learning

Given
Training data annotated with class information.

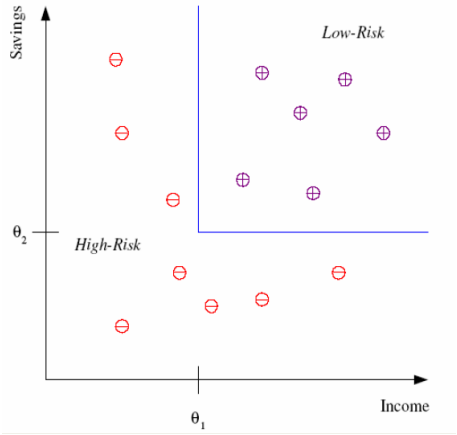
Goal
Build a *model* which will allow classification of new data.

Method



1. Feature extraction: Convert samples into vectors.
2. Training: Automatically learn a model.
3. Classification: Apply the model on new data.

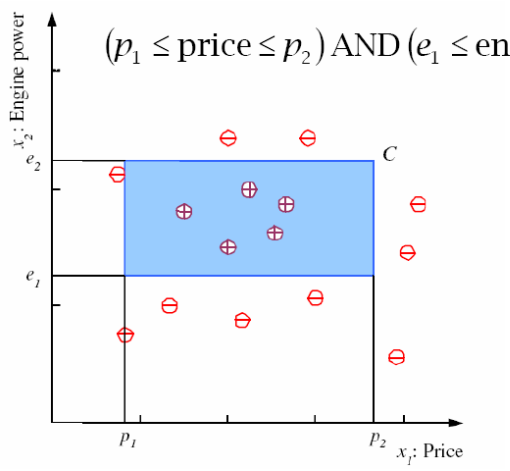
Supervised Learning Example: Bank Customers



(from Alpaydin (2004))

This is an example of a training dataset where each circle corresponds to one data instance with input values in the corresponding axes and its sign indicates the class. For simplicity, only two customer attributes, income and savings, are taken as input and the two classes are low-risk ('+') and high-risk ('−'). An example model that separates the two types of examples is also shown. The model has two parameters, θ_1 and θ_2, which need to be set. These parameters are either set manually by observing the training data, or automatically by training the model using supervised machine learning. Image from: E. Alpaydin. 2004. Introduction to Machine Learning. ©The MIT Press.

Supervised Learning Example: Family Cars

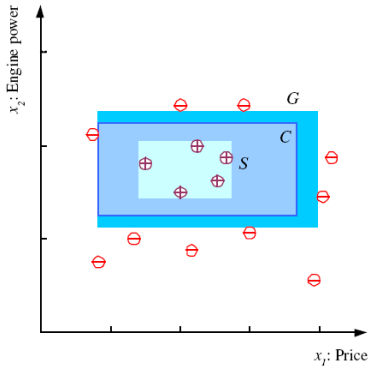


(from Alpaydin (2004))

Another example where now the model has four parameters. The class of family car is a rectangle in the price-engine power space. From: E. Alpaydin. 2004. Introduction to Machine Learning. ©The MIT Press.

Hypothesis Class

- Choice in generality of hypothesis
  - S = Most specific hypothesis: tightest bound
  - G = Most general hypothesis: loosest bound



(from Alpaydin (2004))

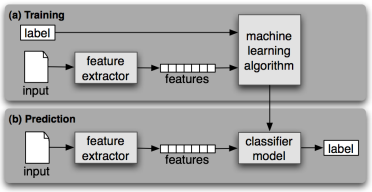
Given the training data, S is the most specific hypothesis that correctly classifies all training data, and G is the most general hypothesis that correctly classifies all training data. In general, the safest hypothesis will be half way between S and G. In this example, C is the actual class. Image from: E. Alpaydin. 2004. Introduction to Machine Learning. ©The MIT Press.

2 Statistical Classification with NLTK and Scikit-Learn

2.1 NLTK

NLTK Features

- Statistical classifiers are not able to make sense of text.
- We need to feed them with our interpretation of the text.
- NLTK classifiers expect a dictionary of features and values.



Example of a Simple Feature Extractor

```
def gender_features(word):
    return {'last_letter': word[-1]}
```

Example: Gender Classification

```
>>> import nltk
>>> from nltk.corpus import names
>>> m = names.words('male.txt')
>>> len(m)
2943
>>> f = names.words('female.txt')
>>> len(f)
5001
>>> import random
>>> random.seed(1234) # Fixed random seed to facilitate replicability
>>> names = ([(name, 'male') for name in m] +
             [(name, 'female') for name in f])
>>> random.shuffle(names)
>>> def gender_features(word):
    return {'last_letter': word[-1]}
>>> featuresets = [(gender_features(n), g) for n, g in names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> classifier.classify(gender_features('Neo'))
'male'
>>> classifier.classify(gender_features('Trinity'))
'female'
>>> nltk.classify.accuracy(classifier, test_set)
0.776
>>>
```

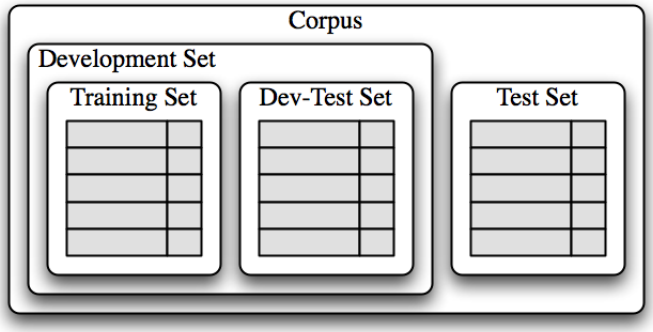
Note that the classifier is fed with the gender features and not with the actual names.

The Development Set

Important
Always test your system with data that has not been used for development (Why ...?)

Development and Test Sets

- Put aside a test set and don't even look at its contents.
- Use the remaining data as a development set.
  - Separate the development set into training and dev-test sets.
  - Use the training set to train the statistical classifiers.
  - Use the dev-test set to fine-tune the classifiers and do error analysis.



Error Analysis in our Gender Classifier

```
>>> train_names = names[1500:]
>>> devtest_names = names[500:1500]
>>> test_names = names[:500]
>>> train_set = [(gender_features(n), g) for n, g in train_names]
>>> devtest_set = [(gender_features(n), g) for  n, g in devtest_names]
>>> test_set = [(gender_features(n), g) for n, g in test_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, devtest_set)
0.752
>>> errors = []
>>> for name, tag in devtest_names:
        guess = classifier.classify(gender_features(name))
        if guess != tag:
            errors.append((tag, guess, name))

>>> len(errors)
248
>>> for (tag,guess,name) in sorted(errors):
      print("correct=%-8s guess=%-8s name=%-30s" % (tag,guess,name))
```

A Revised Gender Classifier

```
>>> def gender_features2(word):
      return {'suffix1': word[-1:],
              'suffix2': word[-2:]}

>>> train_set2 = [(gender_features2(n), g) for n, g in train_names]
>>> devtest_set2 = [(gender_features2(n),g) for n, g in devtest_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set2)
```

```
>>> nltk.classify.accuracy(classifier, devtest_set2)
0.77
```

Beware of Over-fitting

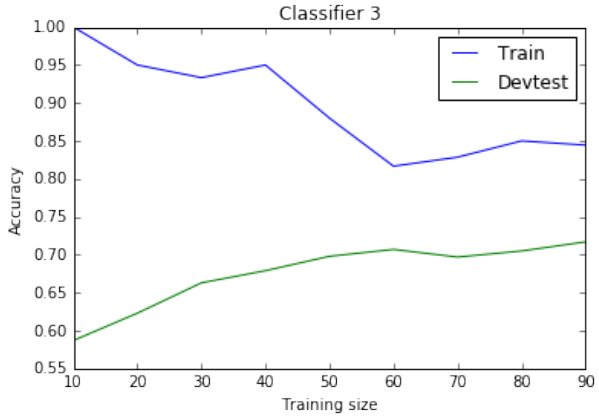
If there are many features on a small corpus the system may *over-fit*.

```
>>> def gender_features3(name):
        features = {}
        features['firstletter'] = name[0].lower()
        features['lastletter'] = name[-1]
        for letter in 'abcdefghijklmnopqrstuvwxyz':
            features['count(%s)' % letter] = name.lower().count(letter)
            features['has(%s)' % letter] = (letter in name.lower())
        return features

>>> gender_features3('John')
{'count(u)': 0, 'has(d)': False, 'count(b)': 0, 'count(w)': 0, ...}
>>> train_set3 = [(gender_features3(n), g) for n, g in train_names]
>>> devtest_set3 = [(gender_features3(n), g) for n, g in devtest_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set3[:50])
>>> nltk.classify.accuracy(classifier, devtest_set3)
0.698
>>> classifier2b = nltk.NaiveBayesClassifier.train(train_set2[:50])
>>> nltk.classify.accuracy(classifier2b, devtest_set2)
0.7
```

Some types of classifiers are more sensitive to over-fitting than others.

Identifying Over-fitting



We can observe over-fitting when the evaluation on the training set is much better than that on the test set. Over-fitting is generally lesser as we increase the size of the training set (X axis). In this example, the system overfits in all cases. Overfitting has reduced as we increase the number of training samples but there is still substantial overfitting even with 90 training samples.

We can also see that accuracy of the test set keeps increasing as we increase the training size. This probably means that accuracy of the test set will probably increase even further so we probably want to add more training data. Try it out yourself! What happens if you train classifier3 with the entire training data? Is there a point where adding more training data does not help?

2.2 Scikit-Learn

Text Classification in Scikit-Learn

- Scikit-learn includes a large number of statistical classifiers.
- All of the classifiers have a common interface.
- The features of a document set are represented as a matrix.
  - Each row represents a document.
  - Each column represents a feature.
- Scikit-learn provides some useful feature extractors for text:
  1. CountVectorizer returns a (sparse) matrix of word counts.
  2. TfidfVectorizer returns a (sparse) matrix of tf.idf values.

Gender Classifier in Scikit-Learn - Take 1

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> def gender_features(word):
        "Return the ASCII value of the last two characters"
        return [ord(word[-2]), ord(word[-1])]
>>> featuresets = [(gender_features(n), g) for n, g in names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> train_X, train_y = zip(*train_set)
>>> classifier = MultinomialNB()
>>> classifier.fit(train_X, train_y)
>>> test_X, test_y = zip(*test_set)
>>> classifier.predict(test_X[:5])
array(['female', 'female', 'male', 'female', 'female'],
      dtype='|S6')
```

In this code, the function "gender_features" returns a list with the ASCII values of the last two characters of the word. This is so because the matrix can only have numerical values.

We also see Python's way to unpack a list. The variable "train_set" is a list where each element is a pair (vector, outcome). We want to unpack the list into two lists: one containing all the vectors, and the other containing all the outcomes. The instruction:

```
train_X, train_y = zip(*train_set)
```

is equivalent to this sequence of instructions:

```
train_X = [x[0] for x in train_set]
train_y = [x[1] for x in train_set]
```

That is, assign to "train_X" the list of all training vectors, and assign to "train_y" the list of all target outcomes. The "*" in the instruction is important.

The complete code of a working program is:

```
from nltk.corpus import names
from sklearn.naive_bayes import MultinomialNB

def gender_features1(word):
```

```
    return [ord(word[-1])]

def gender_features2(word):
    return [ord(word[-1]), ord(word[-2])]

def gender_features3(name):
    features = []
    features.append(ord(name[0].lower()))
    features.append(ord(name[-1]))
    for letter in 'abcdefghijklmnopqrstuvwxyz':
        features.append(name.lower().count(letter))
        features.append(int(letter in name.lower()))
    return features

def accuracy(classifier,test_X,test_y):
    "Return the classifier accuracy"
    results = classifier.predict(test_X)
    correct = results[results == test_y] # returns the list of results that
                                         # are correctly predicted
    return float(len(correct))/len(test_y)

def demo(data,featurefunction):
    featuresets = [(featurefunction(n),g) for (n,g) in data]
    train_set,test_set = featuresets[500:], featuresets[:500]
    train_X,train_y = zip(*train_set)
    test_X,test_y = zip(*test_set)
    classifier = MultinomialNB()
    classifier.fit(train_X,train_y)
    print "Classification of Neo:", \
        classifier.predict([featurefunction('Neo')])[0]
    print "Classification of Trinity:", \
        classifier.predict([featurefunction('Trinity')])[0]
    print "Classifier accuracy:", accuracy(classifier,test_X,test_y)


if __name__ == "__main__":
    m = names.words('male.txt')
    print "There are", len(m), "samples of male names"

    f = names.words('female.txt')
    print "There are", len(f), "samples of female names"

    import random
    names = ([(name,'male') for name in m] +
             [(name,'female') for name in f])
    random.shuffle(names)

    print "Classifier 1:"
    demo(names,gender_features1)
    print
```

```
print "Classifier 2:"
demo(names, gender_features2)
print

print "Classifier 3:"
demo(names, gender_features3)
```

Gender Classification - Take 2

- In the past slide we have used this code to encode the last two characters of a name:

```
def gender_features(word):
    "Return the ASCII value of the last two characters"
    return [ord(word[-2]), ord(word[-1])]
```

- This code is not entirely correct since it is representing characters as numbers.
- In general, non-numerical information is best represented using one-hot encoding.
- sklearn provides the following functions to produce one-hot-encoding vectors:
  - preprocessing.OneHotEncoding: from integers to one-hot vectors.
  - preprocessing.LabelBinarizer: from labels to one-hot vectors.

One-hot Encoding

- Suppose you want to encode five labels: 'a', 'b' ,'c', 'd', 'e'.
- Each label represents one element in the one-hot vector.
- Thus:
  - 'a' is represented as (1, 0, 0, 0, 0).
  - 'b' is represented as (0, 1, 0 ,0 ,0).
  - and so on.
- This is also called binarization or categorical encoding.

One-hot Encoding for Gender Classification

```
def one_hot_character(c):
  alphabet = 'abcdefghijklmnopqrstuvwxyz'
  result = [0]*(len(alphabet)+1)
  i = alphabet.find(c.lower())
  if i >= 0:
      result[i] = 1
  else:
      result[len(alphabet)] = 1 # out of the alphabet
  return result

def gender_features(word):
    last = one_hot_character(word[-1])
    secondlast = one_hot_character(word[-2])
    return secondlast + last
```

3 Advice on Machine Learning

Possible Problems with Machine Learning

- ML methods are typically seen as black boxes.
- Some methods are better than others for specific tasks but people tend to just try several and choose the one with best results.

Possible problems/mistakes you might face

1. Train and test are the same dataset (don't do this!).
2. The results on the test set are much worse than on the dev-test set.
3. The results of both the test set ant the training set are bad.
4. The train/test partition is not random.
5. The results on the test set are good but then the results on your real application problem are bad.

Partition into Training and Testing Set

What's wrong with this partition?

```
from nltk.corpus import names
m = names.words('male.txt')
f = names.words('female.txt')
names = ([(name, 'm') for name in m]+
         [(name, 'f') for name in f])
trainset = names[500:]
testset = names[:500]
```

Advice

1. Make sure that the train and test sets have no bias.
2. Make sure that the train ant test sets are representative of your problem.

Randomised Partition

A better partition

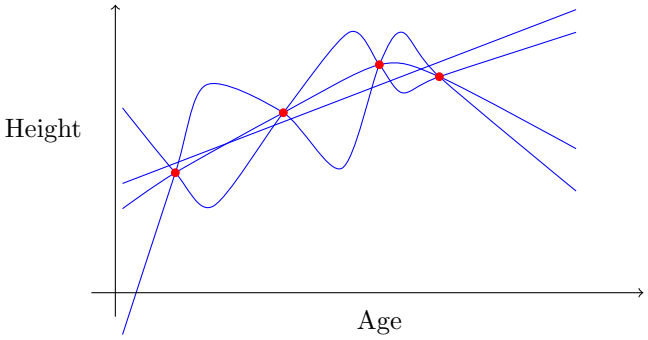
```
from nltk.corpus import names
import random
m = names.words('male.txt')
f = names.words('female.txt')
names = ([(name, 'm') for name in m]+
         [(name, 'f') for name in f])
random.seed(1234)
random.shuffle(names)
trainset = names[500:]
testset = names[:500]
```

When you create a random partition it is useful that different runs of the same code generates the same partitions. For this reason, we set a fixed random seed. The choice of number does not matter (1234 in our case), as long as it is the same seed number in each run.

Unfortunately the random seeds do not guarantee the same generation of random numbers in different machines because the random generation algorithm is implementation-specific.

3.1 Over-fitting

Why is machine learning hard?



- There are an infinite number of curves that fit the data
  - even more if we don't require the curves to exactly fit (e.g., if we assume there's noise in our data).
- In general, more data would help us identify the correct curve better.

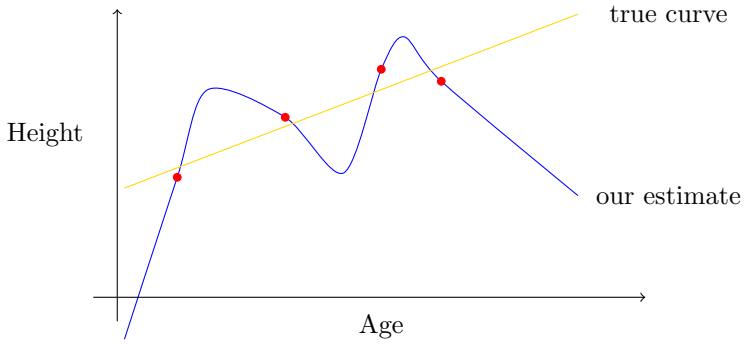
The "no free lunch theorem"

- The **"no free lunch theorem"** says there is no single best way to generalise that will be correct in all cases.

  $\Longrightarrow$ a machine learning algorithm that does well on some problems will do badly on others.

  $\Longrightarrow$ balancing the trade-off between the fit to data and model complexity is a central theme in machine learning.
- Even so, in practice there are machine learning algorithms that do well on broad classes of problems.
- But it's important to understand the problem you are trying to solve as well as possible.

Over-fitting the Training Data



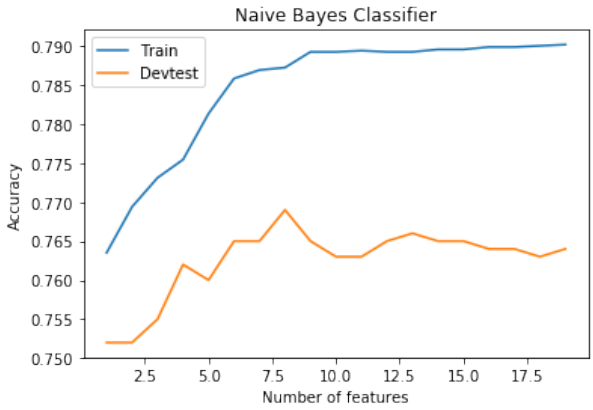
- Over-fitting occurs when an algorithm learns a function that is fitting noise in the data.
- Diagnostic of over-fitting: performance on training data is much higher than performance on dev or test data.

3.2 The Dev-test Set

Fine-tuning the System

- Many ML algorithms have parameters that adjust the tendency to learn from the data.
- A reasonable approach to find the best parameters is to try many options and select the best one.
- For example, we may want to determine the optimal number of features in our classifier.

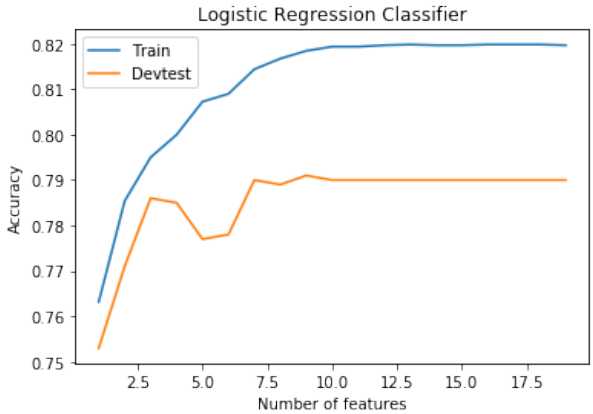
Example of the Impact of Features with the Names Corpus



See related notebook for the code.

This plot shows the evaluation results of the training and dev-test sets as we change the number of characters to consider as features to represent the name (from 1 to 19 characters). We observe that the training set improves as we increase the number of characters. This is expected since we have more information. But overfitting increases slightly, and the results of the devtest set deteriorate when we use more than 8 characters. We should then select 8 characters as the optimal set of features.

Example of the Impact of Features with the Names Corpus



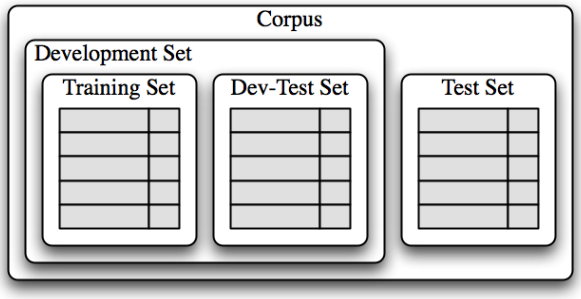
See related notebook for the code.

We see that logistic regression behaves differently, and any number of features larger than 7 will give optimal results. We also see that logistic regression performs better than Naive Bayes. The results may vary with other applications and with other data sets, but in general logistic regression often performs better than Naive Bayes, especially if we have enough training data. Also, logistic regression often is very robust and results do not deteriorate (much) as we increase the number of features.

As we will see in next lecture, logistic regression is actually equivalent to one of the simplest neural networks that you can build.

Introducing the Dev-test Set

- You need to use data that is not in the training set nor in the test set to fine-tune the system.
- It is therefore advisable to have three partitions.
- The test set is used only for the final test, not for fine-tuning parameters.



Take-home Messages

1. Explain and demonstrate the need for separate training, dev-test, and test set.
2. Implement feature extractors for statistical classifiers in NLTK and Scikit-Learn.
3. Use NLTK's and Scikit-Learn's statistical classifiers.
4. Detect over-fitting.

What's Next

Week 4

- Deep Learning.

Reading

- Deep Learning book chapters 2 and 3, and Chapter 6 Section 1.