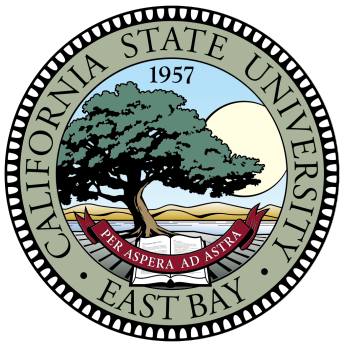


CS 6820 – Machine Learning

Lecture 8

Instructor: Eric S. Gayles, PhD.

Jan 30, 2018

Quick Aside

Monte Carlo Methods

- Monte Carlo methods (or Monte Carlo experiments) are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results.

- They use randomness to solve problems that might be deterministic in principle.

- Monte Carlo methods are mainly used in three distinct problem classes: optimization, numerical integration, and generating draws from a probability distribution.

Monte Carlo Methods

- In principle, Monte Carlo methods can be used to solve any problem having a probabilistic interpretation.

- By the law of large numbers, integrals described by the expected value of some random variable can be approximated by taking the empirical mean (a.k.a. the sample mean) of independent samples of the variable.

Estimating Sums with Samples

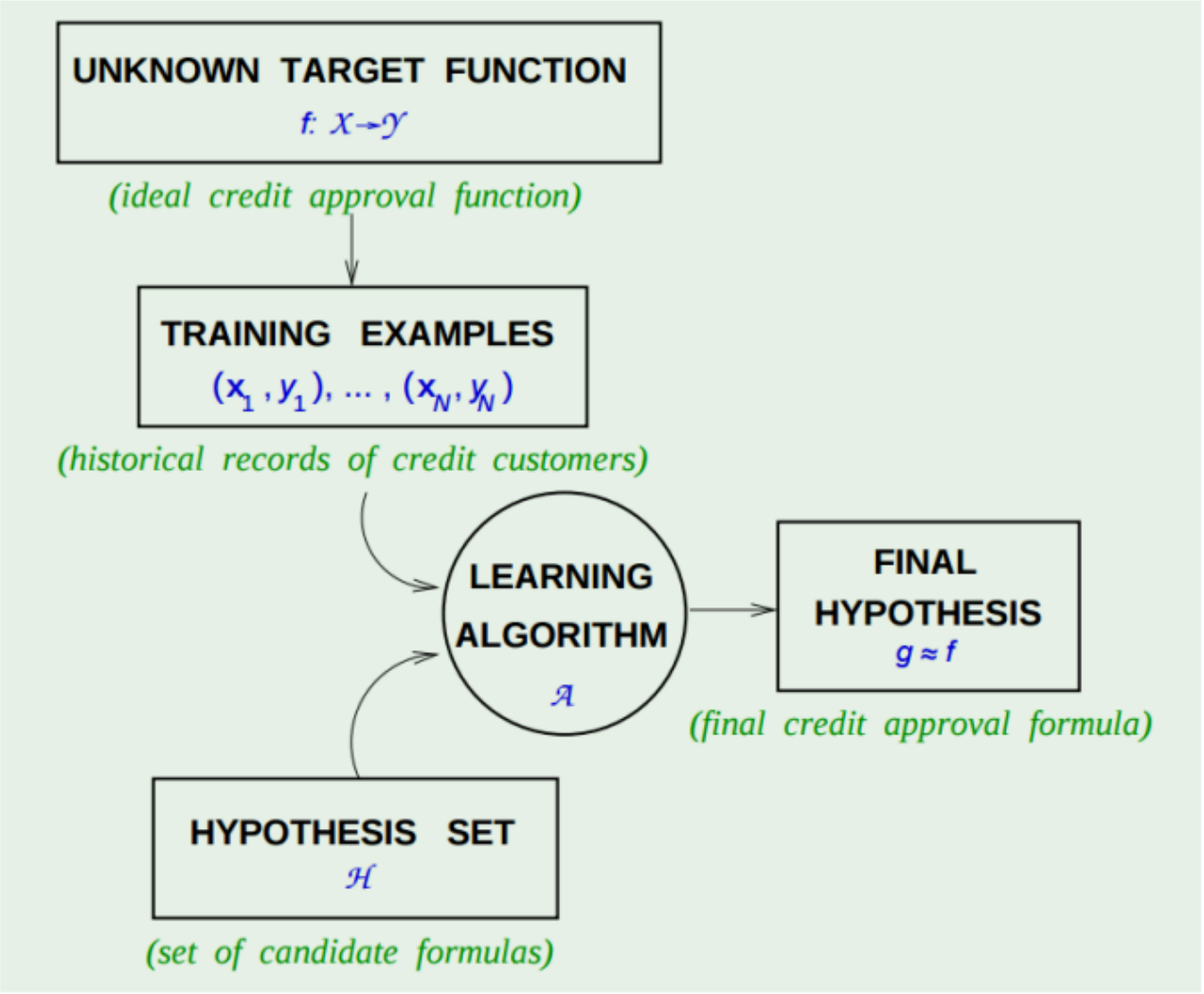
$$s = \sum_{\boldsymbol{x}} p(\boldsymbol{x}) f(\boldsymbol{x}) = E_p[f(\mathbf{x})] \qquad (17.1)$$

$$s = \int p(\boldsymbol{x}) f(\boldsymbol{x}) d\boldsymbol{x} = E_p[f(\mathbf{x})]$$

$$\hat{s}_n = \frac{1}{n} \sum_{i=1}^{n} f(\boldsymbol{x}^{(i)}).$$

End Aside

Hypothesis Development



BTW - Matplotlib Colors

Commands which take color arguments can use several formats to specify the colors. For the basic built-in colors, you can use a single letter

- `b`: blue
- `g`: green
- `r`: red
- `c`: cyan
- `m`: magenta
- `y`: yellow
- `k`: black
- `w`: white

To use the colors that are part of the active color cycle in the current style, use `C` followed by a digit. For example:

- `C0`: The first color in the cycle
- `C1`: The second color in the cycle

Gray shades can be given as a string encoding a float in the 0-1 range, e.g.:

```
color = '0.75'
```

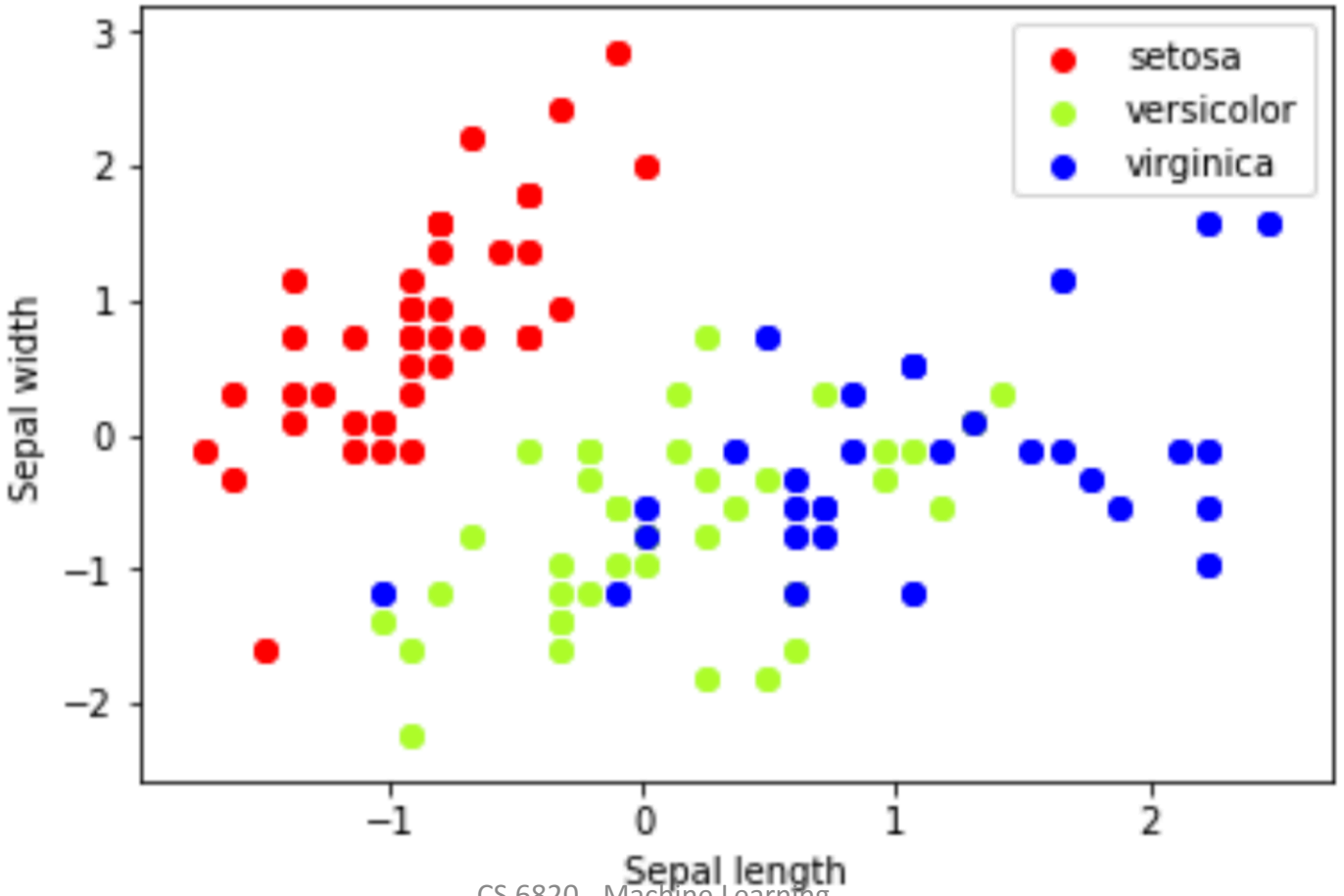
For a greater range of colors, you have two options. You can specify the color using an html hex string, as in:

```
color = '#eeefff'
```

In [9]:
```
from sklearn.cross_validation import train_test_split
from sklearn import preprocessing
import matplotlib.pyplot as plt

#Get dataset with only the first two attributes
from sklearn import datasets
iris = datasets.load_iris()
X_iris, y_iris = iris.data, iris.target
X, y = X_iris[:, :2], y_iris

# Split the dataset into a training and a testing set
# Test set will be the 25% taken randomly
#     NOTE: If random_state is None or np.random, then a randomly-initialized RandomState
#           object is returned.
#
# If random_state is an integer, then it is used to seed a new RandomState object.
#
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=33)
print (X_train.shape, y_train.shape)

# Standardize the features
scaler = preprocessing.StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Plot the results
colors = ['red', 'greenyellow', 'blue']
for i in range(len(colors)):
    xs = X_train[:, 0][y_train == i]
    ys = X_train[:, 1][y_train == i]
    plt.scatter(xs, ys, c=colors[i])
    plt.legend(iris.target_names)
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
plt.show()
```

Classification with Linear Regression

```
(112, 2) (112,)
```



Classification with Linear Regression

- From the picture, it seems that we could draw straight lines (or Hyperplanes) that correctly separates the sets.

- Defines our decision boundary – meaning class membership depends on what side of the hyperplane the instance is.

Classification with Linear Regression

- To implement a linear classification, we could will use the SGDClassifier from scikit-learn.
- SGD stands for Stochastic Gradient Descent
- The algorithm will learn the coefficients of the hyperplane by minimizing the loss function.

In [64]:
```
from sklearn import linear_model
import numpy as np
import warnings
warnings.filterwarnings(action='ignore', category=FutureWarning)

#Train the dataset
clf = linear_model.SGDClassifier()
clf.fit(X_train, y_train)
```

Out[64]:
```
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)
```

In [65]:
```
# Pring the solution
print (clf.coef_)
print (clf.intercept_)
```

```
[[-32.28900526  17.78204246]
 [ -3.84646435 -18.24234056]
 [  9.5695774   -3.31899158]]
[-18.22958627  -9.00109423 -10.70472659]
```

Classification with Linear Regression

- But, why does our coefficient matrix have three rows?

- Because we did not tell the method that we have are facing a three-class problem, not a binary decision problem.

- What the classifier did was to convert the problem into three binary classification problems in a one-versus-all setting (it proposes three lines that separate a class from the rest).

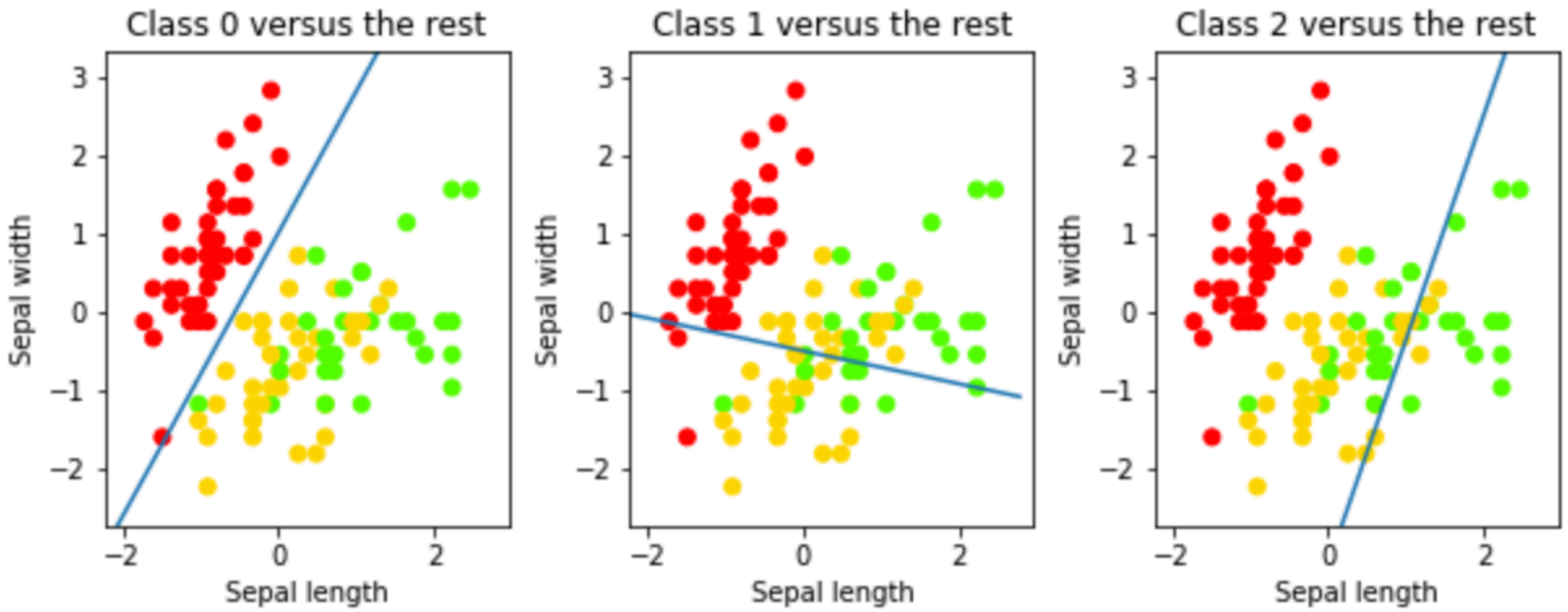
Classification with Linear Regression

- We are predicting a class from the possible three classes.
- … but linear models are essentially binary: something is missing.
- Our prediction procedure combines the result of the three binary classifiers and selects the class in which it is more confident.

Classification with Linear Regression

In [66]:
```
x_min, x_max = X_train[:, 0].min() - .5, X_train[:, 0].max() + .5
y_min, y_max = X_train[:, 1].min() - .5, X_train[:, 1].max() + .5
xs = np.arange(x_min, x_max, 0.5)
fig, axes = plt.subplots(1, 3)
fig.set_size_inches(10, 6)
for i in [0, 1, 2]:
    axes[i].set_aspect('equal')
    axes[i].set_title('Class '+ str(i) + ' versus the rest')
    axes[i].set_xlabel('Sepal length')
    axes[i].set_ylabel('Sepal width')
    axes[i].set_xlim(x_min, x_max)
    axes[i].set_ylim(y_min, y_max)
    plt.sca(axes[i])
    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.prism)
    ys = (-clf.intercept_[i] - xs * clf.coef_[i, 0]) / clf.coef_[i, 1]
    plt.plot(xs, ys, hold=True)
plt.show()
```

Classification with Linear Regression



Moving on

Feature Extraction – One Hot Encoding

```
In [5]: from sklearn.feature_extraction import DictVectorizer
        onehot_encoder = DictVectorizer()
        instances = [  {'city': 'New York'},
                       {'city': 'San Francisco'},
                       {'city': 'Chapel Hill'}]
        print (onehot_encoder.fit_transform(instances).toarray())

        [[ 0.  1.  0.]
         [ 0.  0.  1.]
         [ 1.  0.  0.]]

In [ ]:
```

- Many machine learning problems use text as an explanatory variable.

- Text must be transformed into a different representation that encodes as much of its meaning as possible in a feature vector.

* Example and comments from *Mastering Machine Learning with Scikit-Learn*, Hackeling

Feature Extraction - Text

- The most common representation of text is the **bag-of-words** model.
- This representation uses a multiset, or bag, that encodes the words that appear in a text
- The bag-of-words does not encode any of the text's syntax, ignores the order of words, and disregards all grammar.
- Bag-of-words can be thought of as an extension to one-hot encoding. It creates one feature for each word of interest in the text.
- The bag-of-words model is motivated by the intuition that documents containing similar words often have similar meanings.
- The bag-of-words model can be used effectively for document classification and retrieval despite the limited information that it encodes.

Feature Extraction - Text

- Our corpus has eight unique words, so each document will be represented by a vector with eight elements.
- The number of elements that comprise a feature vector is called the vector's **dimension**
- A **dictionary** maps the vocabulary to indices in the feature vector.

```
In [6]: corpus = [
            'UNC played Duke in basketball',
            'Duke lost the basketball game']
```

Feature Extraction - Text

- In the most basic bag-of-words representation, each element in the feature vector is a binary value that represents whether or not the corresponding word appeared in the document.
- The CountVectorizerclass can produce a bag-of-words representation from a string or file.
- By default, CountVectorizer converts the characters in the documents to lowercase, and **tokenizes** the documents.
- Tokenization is the process of splitting a string into **tokens**, or meaningful sequences of characters.
- Tokens frequently are words, but they may also be shorter sequences including punctuation characters, etc.
- The CountVectorizer class tokenizes using a regular expression that splits strings on whitespace and extracts sequences of characters that are two or more characters in length.

Feature Extraction - Text

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
print (vectorizer.fit_transform(corpus).todense())
print (vectorizer.vocabulary_)
```

```
[[1 1 0 1 0 1 0 1]
 [1 1 1 0 1 0 1 0]]
{'unc': 7, 'played': 5, 'duke': 1, 'in': 3, 'basketball': 0, 'lost': 4, 'the': 6, 'game': 2}
```

Feature Extraction - Text

- Let's add a third sentence to our Corpus.

```
corpus = [
    'UNC played Duke in basketball',
    'Duke lost the basketball game',
    'I at a sandwitch']
```

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
print (vectorizer.fit_transform(corpus).todense())
print (vectorizer.vocabulary_)
```

```
[[0 1 1 0 1 0 1 0 0 1]
 [0 1 1 1 0 1 0 0 1 0]
 [1 0 0 0 0 0 0 1 0 0]]
{'unc': 9, 'played': 6, 'duke': 2, 'in': 4, 'basketball': 1, 'lost': 5, 'the': 8, 'game': 3, 'at': 0, 'sandwitch': 7}
```

Feature Extraction - Text

The meanings of the first two documents are more similar to each other than they are to the third document, and their corresponding feature vectors are more similar to each other than they are to the third document's feature vector when using a metric such as **Euclidean distance**. The Euclidean distance between two vectors is equal to the **Euclidean norm**, or L2 norm, of the difference between the two vectors:

$$d = \|x_0 - x_1\|$$

This is essentially the distance between two points:

$$\mathrm{d}(\mathbf{p}, \mathbf{q}) = \mathrm{d}(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

Recall that the Euclidean norm of a vector is equal to the vector's magnitude, which is given by the following equation:

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$$

Feature Extraction - Text

```
corpus = [
    'UNC played Duke in basketball',
    'Duke lost the basketball game',
    'I at a sandwitch']
```

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
print (vectorizer.fit_transform(corpus).todense())
print (vectorizer.vocabulary_)
```

```
[[0 1 1 0 1 0 1 0 0 1]
 [0 1 1 1 0 1 0 0 1 0]
 [1 0 0 0 0 0 0 1 0 0]]
{'unc': 9, 'played': 6, 'duke': 2, 'in': 4, 'basketball': 1, 'lost': 5, 'the': 8, 'game': 3, 'at': 0, 'sandwitch': 7}
```

```
from sklearn.metrics.pairwise import euclidean_distances
counts = [
        [0, 1, 1, 0, 0, 1, 0, 1],
        [0, 1, 1, 1, 1, 0, 0, 0],
        [1, 0, 0, 0, 0, 0, 1, 0]]
euclidean_distances(counts)
```

```
array([[ 0.        ,  2.        ,  2.44948974],
       [ 2.        ,  0.        ,  2.44948974],
       [ 2.44948974,  2.44948974,  0.        ]])
```