An Introduction to Multi-label Learning
(ML-KNN & BP-MLL)

Bobby Lumpkin

Overview

1. Introduction to Multi-label Learning
   - Overview and Advantages
2. ML-KNN Approach
   - Model Outline
   - Computing Model Probabilities
   - Implementation (in scikit-multilearn)
3. BP-MLL Approach
   - Feed-forward Neural Networks
   - Neural Network Loss Functions & Training for MLL
   - Implementation (in TensorFlow/Keras)

Introduction to Multi-label Learning

Multi-label Learning

What is it?

- → Multi-label learning (mll) is a form of classification where each instance may be associated with more than one label.
- → Plenty of tasks, such as text categorization, functional genomics, and supervised product recommendation fit naturally into the mll paradigm.
  - * **EX:** A news article discussing White House Covid press briefings might belong to each of the categories: "News", "Health" and "Government".

Why use novel approaches?

- → Naive Approach: Train a sequence of independent binary classifiers (one per category)
- → Doesn't capitalize on the information in the correlations between the different labels of each instance.

Multi-label Paradigm: Definitions & Notation

- Let χ denote the domain of instances and $\mathcal{Y} = \{1, ..., Q\}$ be the finite set of labels.
- Given $x \in \chi$ and its associated $Y \subseteq \mathcal{Y}$, let $\vec{y}_x$ be the category vector for x such that (for all $\ell \in \mathcal{Y}$) $\vec{y}_x(\ell) = 1$ if $\ell \in Y$. Otherwise, $\vec{y}_x(\ell) = 0$.

ML-KNN Approach

ML-KNN Algorithm: More Notation

Notation:

- Let $N(x)$ denote the set of K nearest neighbors of x, identified in the training set.
- Let $\vec{C}_x(\ell) = \sum_{a \in N(x)} \vec{y}_a(\ell)$ $(\ell \in \mathcal{Y})$ define a membership counting vector.
- Let H_0^ℓ denote the event that test instance t does not have a label ℓ and let H_1^ℓ denote the event that it does have label ℓ.
- Let E_j^ℓ $(j \in \{1, ..., K\})$ denote the event that, among the K nearest neighbors of t, there are exactly j instances which have label ℓ.

ML-KNN Algorithm: Overall Approach

Overall Approach: This ML-KNN algorithm takes a parametric, Bayesian approach towards estimating the Bayes Optimal Classifier. As with the single-label algorithm, it does this using the K nearest neighbors of an instance. Namely...

- Given a test instance, t, $\vec{Y}_t$ is determined using the MAP estimate:

$$\begin{aligned}
\vec{y}_t(\ell) &= \underset{b\in\{0,1\}}{\operatorname{argmax}} \, \mathbb{P}\left(\mathrm{H}_b^\ell | E^\ell_{\vec{C}_t(\ell)}\right), \quad \ell \in \mathcal{Y} \\
&= \underset{b\in\{0,1\}}{\operatorname{argmax}} \frac{\mathbb{P}\left(\mathrm{H}_b^\ell\right) \cdot \mathbb{P}\left(E^\ell_{\vec{C}_{t(\ell)}} | \mathrm{H}_b^\ell\right)}{\mathbb{P}\left(E^\ell_{\vec{C}_t(\ell)}\right)} \\
&= \underset{b\in\{0,1\}}{\operatorname{argmax}} \, \mathbb{P}\left(\mathrm{H}_b^\ell\right) \cdot \mathbb{P}\left(E^\ell_{\vec{C}_{t(\ell)}} | \mathrm{H}_b^\ell\right)
\end{aligned}$$

- Where we take a Bayesian approach towards estimating the prior probabilities, $\mathbb{P}\left(\mathrm{H}_b^\ell\right)$, and conditional probabilities, $\mathbb{P}\left(E^\ell_{\vec{C}_{t(\ell)}} | \mathrm{H}_b^\ell\right)$.

ML-KNN Algorithm: Overall Approach continued...

Definition: Let $\vec{\mathrm{r}}_t$ denote the real-valued vector with ℓ^{th} component:

$$\vec{\mathrm{r}}_t(\ell) := \mathbb{P}\left(\mathrm{H}_1^\ell\right).$$

$\Rightarrow$ Thus, given training data, $\mathcal{X}$, and a test instance, t, we wish to compute $[\vec{y}_t(\ell), \vec{r}_t(\ell)]$.

ML-KNN Algorithm: Computing the Prior Probabilities, $\widehat{\mathbb{P}(\mathrm{H}_b^\ell)}$

We model $\mathbb{P}(\mathrm{H}_1^\ell)$ with a $\mathrm{Beta}(s,s)$ prior and $\mathrm{Bernoulli}(\mathbb{P}(\mathrm{H}_1^\ell))$ likelihood. (When $s=1$, $\mathrm{Beta}(s,s)$ reduces to the uniform distribution on $[0,1]$.)

$\Rightarrow$ The posterior distribution for $\mathbb{P}(\mathrm{H}_1^\ell)$ is:

$$\mathrm{Beta}\left(s+\sum_{i=1}^{m}\vec{y}_{x_i}(\ell),\ s+m-\sum_{i=1}^{m}\vec{y}_{x_i}(\ell)\right).$$

$\Rightarrow$ We will estimate $\mathbb{P}(\mathrm{H}_1^\ell)$ with the expectation of it's posterior Beta distribution:

$$\widehat{\mathbb{P}(\mathrm{H}_1^\ell)} := \frac{s+\sum_{i=1}^{m}\vec{y}_{x_i}(\ell)}{2s+m}$$

where m is the number of training instances.

$\Rightarrow$ We estimate $\widehat{\mathbb{P}(\mathrm{H}_0^\ell)} := 1-\widehat{\mathbb{P}(\mathrm{H}_1^\ell)}$.

ML-KNN Algorithm: Computing the Conditional Probabilities, $\widehat{\mathbb{P}(E_j^\ell|\mathrm{H}_b^\ell)}$

Definition:

(i) Let c be a vector of length $K+1$, where $c(j) =$ the number of training instances where $\vec{C}_{x_i}(\ell) = j$ when $\vec{y}_{x_i}(\ell) = 1$.

(ii) Similarly, let c' be a vector of length $K+1$, where $c(j) =$ the number of training instances where $\vec{C}_{x_i}(\ell) = j$ when $\vec{y}_{x_i}(\ell) = 0$.

Model:

* We give $\overrightarrow{\mathbb{P}(E_j^\ell|\mathrm{H}_1^\ell)}$ (and $\overrightarrow{\mathbb{P}(E_j^\ell|\mathrm{H}_0^\ell)}$) a $\mathrm{Dirichlet}(K+1, (s, ..., s))$ prior distribution
* We use a $\mathrm{Categorical}(K+1, (\frac{c(0)}{m}, ..., \frac{c(K)}{m}))$ likelihood (analogously for $\overrightarrow{\mathbb{P}(E_j^\ell|\mathrm{H}_0^\ell)}$).
  - where $\overrightarrow{\mathbb{P}(E_j^\ell|\mathrm{H}_1^\ell)} = (\mathbb{P}(E_1^\ell|\mathrm{H}_1^\ell), ..., \mathbb{P}(E_K^\ell|\mathrm{H}_1^\ell))$ (analogously for $\overrightarrow{\mathbb{P}(E_j^\ell|\mathrm{H}_0^\ell)}$).

ML-KNN Algorithm: Computing the Conditional Probabilities, $\widehat{\mathbb{P}(E_j^\ell|\mathrm{H}_b^\ell)}$ continued...

⇒ The posterior distribution for $\overrightarrow{\mathbb{P}(E_j^\ell|H_1^\ell)}$ is

$$\mathrm{Dirichlet}\Big(K+1, \big(s+c(0), ..., s+c(K)\big)\Big)$$

(and analogously for $\overrightarrow{\mathbb{P}(E_j^\ell|H_0^\ell)}$).

⇒ Given $j \in \{1, ..., K\}$, we estimate $\mathbb{P}(E_j^\ell|H_1^\ell)$ and $\mathbb{P}(E_j^\ell|H_0^\ell)$ with the expectations of their posterior distributions:

$$\mathbb{P}(E_j^\ell|H_1^\ell) := \frac{(s+c(j))}{((K+1)s + \sum_{n=0}^{K} c(n))}$$
$$\mathbb{P}(E_j^\ell|H_0^\ell) := \frac{(s+c'(j))}{((K+1)s + \sum_{n=0}^{K} c'(n))}$$

Computing $\vec{y}_t$ and $\vec{r}_t$

Using our previous derivation:

$$\widehat{\vec{y}_t(\ell)} := \underset{b\in\{0,1\}}{\operatorname{argmax}} \left[\widehat{\mathbb{P}(H_b^\ell)} \cdot \mathbb{P}(\widehat{E^\ell_{\vec{C}_T(\ell)}|}H_b^\ell)\right]$$

AND

$$\widehat{\vec{r}_t(\ell)} := \frac{\widehat{\mathbb{P}(H_1^\ell)} \cdot \mathbb{P}(\widehat{E^\ell_{\vec{C}_T(\ell)}|}H_1^\ell)}{\sum_{b\in\{0,1\}}\left[\mathbb{P}(H_b^\ell) \cdot \mathbb{P}(E^\ell_{\vec{C}_t(\ell)}|H_b^\ell)\right]}$$
$$= \frac{\widehat{\mathbb{P}(H_1^\ell)} \cdot \mathbb{P}(\widehat{E^\ell_{\vec{C}_T(\ell)}|}H_1^\ell)}{\mathbb{P}(E^\ell_{\vec{C}_t(\ell)})}$$

NOTE: The larger the value for s, the less importance assigned to the training data: As $s \to \infty$, $\vec{r}_t(\ell) \to \frac{1}{2}$.

Scikit-multilearn Implementation

Scikit-multilearn:

- The **"scikit-learn"** module is a free and widely used software machine learning library for Python, including many popular regression, classification and unsupervised learning algorithms.
- The **"scikit-multilearn"** module is a library for multi-label classification that is built on top of the scikit-learn ecosystem.

ML-KNN Classification in scikit-multilearn:

→ MLkNN() from the scikit-multilearn module can be used to instantiate a ML-KNN object.

→ "MLkNN" class methods like "fit()" and "predict()" mirror those for standard scikit-learn objects.

```
from skmultilearn.adapt import MLkNN

classifier = MLkNN(k=3)

# train
classifier.fit(X_train, y_train)

# predict
predictions = classifier.predict(X_test)
```

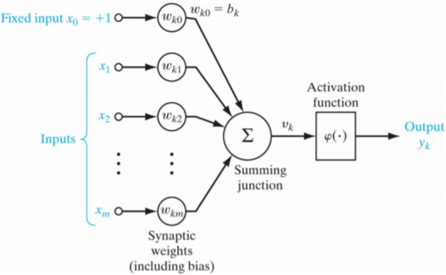
BP-MLL Approach

Introduction to Feed-forward Networks: Perceptron Model

The perceptron model (and feed-forward networks, in general) can be viewed as a connected, directed, loop-free graph like the one below.

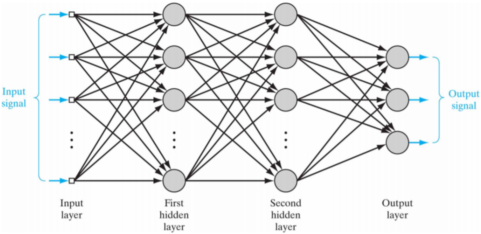
- Neurons in the first layer represent components of the input vectors
- The output of the neuron in the next layer is determined by applying a non-linear "activation function" to a linear combination of the input components, plus a bias.
- Rosenblatt's Perceptron model uses a step function non-linearity, but other common activation functions include the sigmoid function, $\sigma()$, tanh(), ReLU, Leaky ReLU, etc..

$$v_k = \sum_{j=0}^{m} w_{kj} x_j$$



Multilayer Networks & Training

- Adding additional layers and units (like in the network pictured below) significantly expands the class of discrimination problems a network can learn.
- "Online" training involves evaluating an instance, and updating weights using gradient descent.
- For multi-label learning with Q instances, networks will have Q output layers, each with a tanh() activation.



Designing a Cost Function

As for standard networks, BP-MLL uses gradient descent & back propagation for learning. The novelty of the approach is in the design of the cost function.

Naive Approach ("BasicBP"):

- Standard MSE:

$$E = \sum_{i=1}^{m} E_i = \sum_{i=1}^{m} \sum_{j=1}^{q} (c_j^i - d_j^i)^2$$

  where $c_j^i = c_j(x_i)$ is the output of the network on x_i on the j^{th} class.

Novel Approach ("BP-MLL"):

- BP-MLL Cost function:

$$E = \sum_{i=1}^{m} E_i = \sum_{i=1}^{m} \frac{1}{|Y_i||\overline{Y}_i|} \sum_{(k,l) \in Y_i \times \overline{Y}_i} \exp(-(c_k^i - c_l^i))$$

  so that the i^{th} error term is severely penalized if c_k^i is much smaller than c_l^i.

- Back-propagation for training is derived just as in the standard (MSE) case (details omitted here, but can be found in Zhang and Zhou's paper).

Deep Learning APIs in Python: TensorFlow/Keras

TensorFlow: an open source python library for numerical computation and large-scale machine learning, created by the Google Brain team.

→ One of the most widely used APIs for deep learning, along with PyTorch and Keras.

→ Later versions of TensorFlow began incorporating the Keras API, since users found its high-level design to be simpler.

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

TensorFlow Implementation of BP-MLL

BP-MLL in TensorFlow:

→ Data Scientist, Lukas Huwald, published an implementation of the bp-mll cost function for the TensorFlow API as part of the module "bpmll".

→ After installation, the bp-mll loss function can be utilized just as any other TensorFlow loss function.

```
# create simple mlp
model = Sequential()
model.add(Dense(128, input_dim=dim_no, activation='relu', kernel_initializer='glorot_uniform'))
model.add(Dense(64, activation='relu', kernel_initializer='glorot_uniform'))
model.add(Dense(class_no, activation='sigmoid', kernel_initializer='glorot_uniform'))
model.compile(loss=bp_mll_loss, optimizer='adagrad', metrics=[])

# train a few epochs
model.fit(X_train, Y_train, epochs=100)
```

References (Original Method Papers)

Min-Ling Zhang and Zhi-Hua Zhou. Ml-knn: A lazy learning approach to multi-label learning. *Pattern Recognition*, 40(7):2038–2048, 2007. doi: 10.1016/j.patcog.2006.12.019.

Min-Ling Zhang and Zhi-Hua Zhou. Multilabel neural networks with applications to functional genomics and text categorization. *IEEE Transactions on Knowledge and Data Engineering*, 18(10):1338–1351, 2006. doi: doi:10.1109/TKDE.2006.162.