CS 109: Probability for Computer Scientists
Problem Set#6

Adonis Pugh

March 13, 2020

1. We are writing a WebMD program that is slightly larger than the one we worked through in class. In this program we predict whether a user has a flu ($F = 1$) or cold ($C = 1$) based on knowing any subset of 10 potential binary symptoms (e.g., headache, sniffles, fatigue, cough, etc) and a subset of binary risk factors (exposure, stress).

   - We know the prior probability for Stress is 0.5 and Exposure is 0.1.
   - The functions `probCold(s, e)` and `probFlu(s, e)` return the probability that a patient has a cold or flu, given the state of the risk factors stress (`s`) and exposure (`e`).
   - The function `probSymptom(i, f, c)` which returns the probability that the `i`th symptom (X_i) takes on value 1, given the state of cold (`c`) and flu (`f`): $P(X_i = 1|F = f, C = c)$.

   We would like to write pseudocode to calculate the probability of flu *conditioned on observing* that the patient has had exposure to a sick friend and that they are experiencing Symptom 2 (sore throat). In terms of random variables P(Flu = 1 | Exposure = 1, X_2 = 1):

```
def inferProbFlu() # P(Flu = 1 | Exposure = 1 and X_2 = 1)
```

   Write pseudocode that calculates `inferProbFlu()` using **Rejection Sampling**.

Answer.

```
def getSample():
    stress = bernoulli(0.5)
    exposure = bernoulli(0.1)
    cold = bernoulli(probCold(stress, exposure))
    flu = bernoulli(probFlu(stress, exposure))
    symptoms = [None] * 10
    for i in range(len(symptoms)):
        symptoms[i] = bernoulli(probSymptom(i + 1, flu, cold))
 return [stress, exposure, cold, flu, symptoms[i] for i in range(len(symptoms))]

def generateSamples():
```

pset5_webmd.png

```
        N_SAMPLES = 100000
        samples = []
        for i in range(N_SAMPLES):
            sample = getSample()
            samples.append(sample)
        return samples

    def checkConsistent(sample, outcome):
        for i in range(len(outcome)):
            varOutcome = outcome[i]
            varSample = sample[i]
            if varOutcome != None and varOutcome != varSample
                return False
```

```
            return True

        def rejectInconsistent(samples, outcome):
            consistentSamples = []
            for sample in samples:
                if checkConsistent(sample, outcome):
                    consistentSamples.append(sample)
            return consistentSamples

        def rejectionSampling(event, observation):
            samples = generateSamples()
            samplesObservation = rejectInconsistent(samples, observation)
            samplesEvent = rejectInconsistent(samplesObservation, event)
            return len(samplesEvent) / len(samplesObservation)

        def inferProbFlu()
          #for reference: [stress, exposure, cold, flu, symptom[0],...,symptom[9]]
            symptoms = [None] * 10
            symptoms[1] = 1
            observation = [None, 1, None, None, None, ]
           event = [None, None, None, 1, symptom[i] for i in range(len(symptoms))]
            prob = rejectionSampling(event, observation)
```

An alternative approach would be to hard-code this specific event and condition, which doesn't generalize.

```
        def inferProbFlu():
            samples = []
            for i in range(100000):
                stress = bernoulli(0.5)
                exposure = bernoulli(0.1)
                cold = bernoulli(probCold(stress, exposure))
                flu = bernoulli(probFlu(stress, exposure))
                symp_2 = bernoulli(probSymptom(2, flu, cold))
                samples.append[stress, exposure, cold, flu, symp_2]

            outcome = [None, 1, None, None, 1]
            samplesObservation = rejectInconsistent(samples, observation)
            samplesEvent = rejectInconsistent(samplesObservation, event)
            return len(samplesEvent) / len(samplesObservation)

        def rejectInconsistent(samples, outcome):
```

```
            consistentSamples = []
            for sample in samples:
                if checkConsistent(sample, outcome):
                    consistentSamples.append(sample)
            return consistentSamples

        def checkConsistent(sample, outcome):
            for i in range(len(outcome)):
                varOutcome = outcome[i]
                varSample = sample[i]
                if varOutcome != None and varOutcome != varSample
                    return False
            return True
```

2. Consider the Exponential distribution. It is your friend . . . really. Specifically, consider a sample of I.I.D. exponential random variables $X_1, X_2, \ldots, X_n$, where each $X_i \sim \text{Exp}(\lambda)$. Derive the maximum likelihood estimate for the parameter λ in the Exponential distribution.

Answer. The steps here are to (1) define the PDF for our random variables, (2) take the log likelihood the random variables, (3) differentiate the log likelihood, and (4) set the gradient of the log likelihood equal to zero and solve for the parameter(s) of interest.

$$
\begin{aligned}
&X_i \sim Exp(\lambda) \\
&f(X_i \mid \lambda) = \lambda e^{-\lambda X_i} \\
&\theta_{MLE} = \lambda_{MLE} \\
&LL(\theta) = \sum_{i=1}^{n} log(\lambda e^{-\lambda x}) = \sum_{i=1}^{n} [\ln(\lambda) - \lambda X_i] \\
&\frac{\partial LL(\theta)}{\partial \lambda} = \sum_{i=1}^{n} \left[\frac{1}{\lambda} - X_i \right] = 0 \\
&\sum_{i=1}^{n} \frac{1}{\lambda_{MLE}} - \sum_{i=1}^{n} X_i = 0 \\
&\frac{n}{\lambda_{MLE}} = \sum_{i=1}^{n} X_i \\
&\boxed{\lambda_{MLE} = \frac{n}{\sum_{i=1}^{n} X_i}}
\end{aligned}
$$

3. Say you have a set of binary input features/variables $X_1, X_2, \ldots, X_m$ that can be used to make a prediction about a discrete binary output variable Y (i.e., each of the X_i as well as Y can only take on the values 0 or 1). Say that the first k input variables $X_1, X_2, \ldots, X_k$ are actually all identical copies of each other, so that when one has the value 0 or 1, they all do. Explain informally, but precisely, why this may be problematic for the model learned by the Naïve Bayes classifier.

Answer. The core idea that gives the Naïve Bayes classifier algorithm its name is that it assumes variables are conditionally independent of each other given the outcome variable. In this case, that assumption certainly fails because $X_1, X_2, ..., X_k$ are all dependent on each other. The resulting classifier would likely be very inaccurate in handling this set of random variables.

4. Implement a Naïve Bayes classifier. Detailed instructions are provided in the comments of the starter code.

    a. **[Coding]** Implement the function `fit` in `naive_bayes.py`.

    b. **[Coding]** Implement the function `predict` in `naive_bayes.py`.

5. Implement a Logistic Regression classifier. Specifically, you should implement the gradient ascent algorithm described in class. Detailed instructions are provided in the comments of the starter code.

    a. **[Coding]** Implement the function `fit` in `logistic_regression.py`.

    b. **[Coding]** Implement the function `predict` in `logistic_regression.py`.