CSE512 Machine Learning

HW #5

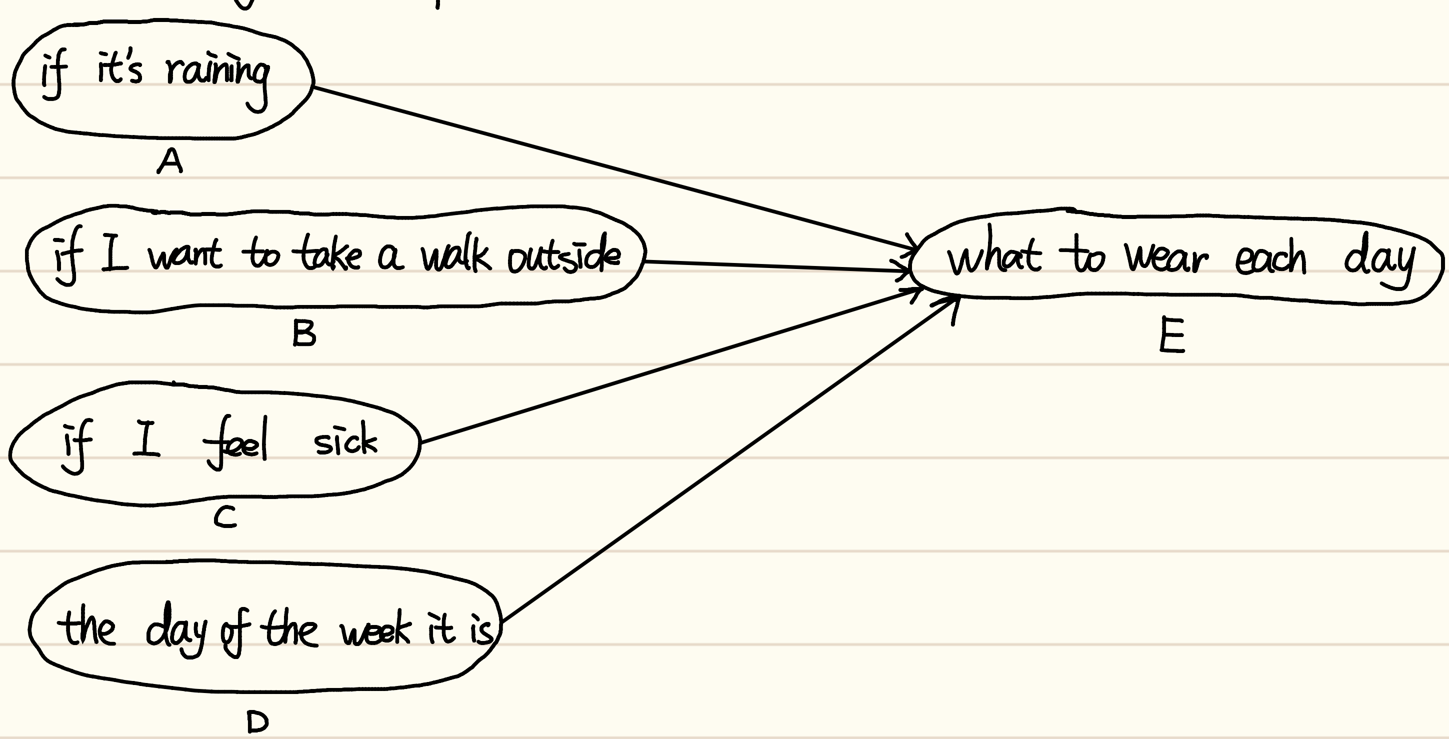
Enbo Yu

113094714

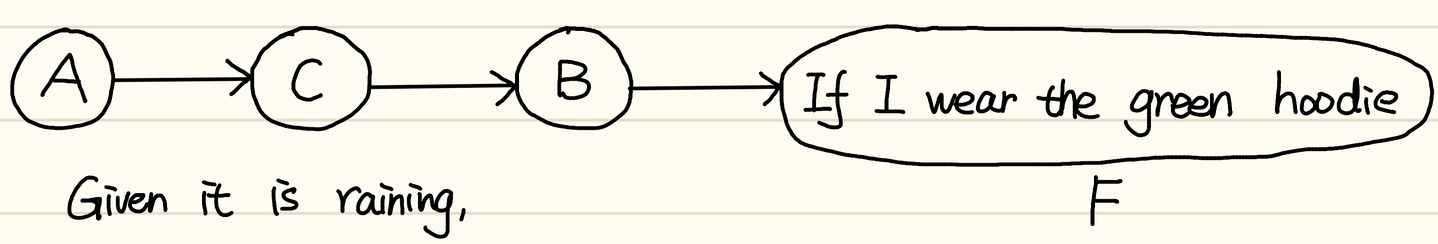
Days extension used (8/8)

1. Directed graphical models and probability inference

(a) Naive Bayes assumption



(b) corresponding graphical mode



Given it is raining,

infer the Pr that I am wearing green hoodie:

Based on the question description, we know that:

- Pr(raining) = 1 ;
- Pr(green hoodie) = Pr(walk);
- Pr(walk | sick) = 0.1, Pr(walk | well) = 0.6;
- Pr(sick | raining) = 0.7;

so $Pr(\text{green hoodie}) = Pr(\text{walk}) = Pr(\text{walk, sick}) + Pr(\text{walk, well})$

$= Pr(\text{walk}|\text{sick}) \cdot Pr(\text{sick}) + Pr(\text{walk}|\text{well}) \cdot Pr(\text{well})$

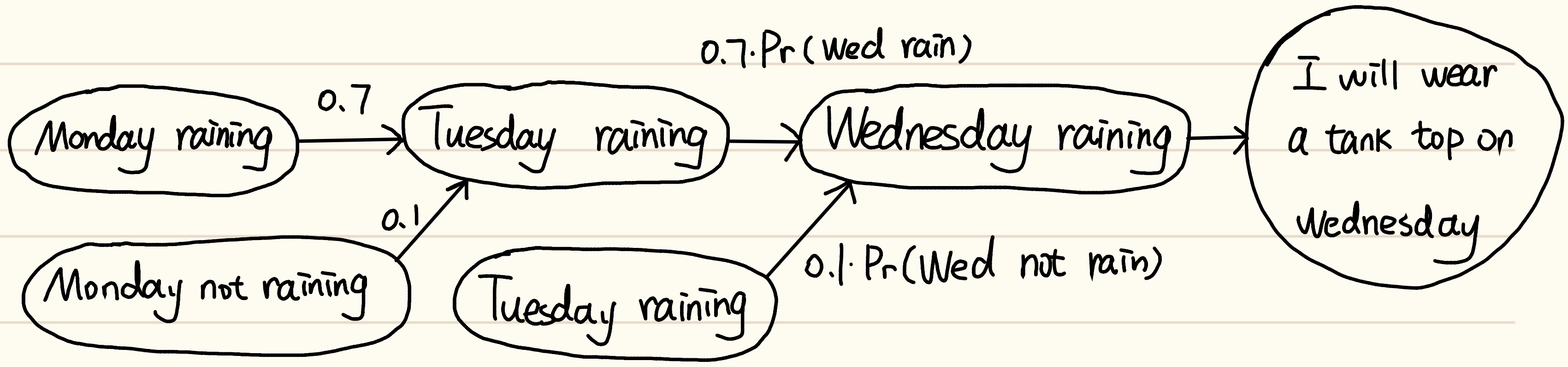
the current condition is under that it is raining, so

$= Pr(\text{walk}|\text{sick}) \cdot Pr(\text{sick}|\text{raining}) + Pr(\text{walk}|\text{well}) \cdot Pr(\text{well}|\text{raining})$

$= 0.1 \cdot 0.7 + 0.6 \cdot (1-0.7) = 0.07 + 0.18 = 0.25.$

So given that it is raining, the probability that I am wearing a green hoodie is 0.25.

(c)



Based on the question description, we know that:

$Pr(\text{Monday raining}) = 1;$

$Pr(\text{cur-day raining}|\text{pre-day raining}) = 0.7$, so $Pr(\text{cur-day not raining}|\text{pre-day raining}) = 1-0.7 = 0.3;$

$Pr(\text{cur-dai raining}|\text{pre-day not raining}) = 0.1$, so $Pr(\text{cur-day not raining}|\text{pre-day not raining}) = 1-0.1 = 0.9.$

so $Pr(\text{Tuesday raining}) = 0.7 \cdot 1 + 0.1 \cdot 0 = 0.7,$

$Pr(\text{Tuesday not raining}) = 1-0.7 = 0.3,$

$Pr(\text{Wednesday raining}) = 0.7 \cdot 0.7 + 0.3 \cdot 0.1 = 0.52.$

$Pr(\text{Wednesday not raining}) = 1 - 0.52 = 0.48.$

So $Pr(\text{Wednesday I wear tank top}) = 0.52 \cdot 0.75 + 0.48 \cdot 0.25 = 0.51$

So the probability that I will wear a tank top on Wednesday is 0.51.

2. Clustering.

- Euclidean distance

```
[8] def get_distance(X,z,D1=None):
        return np.sum(np.power(X - z, 2), axis=1)

    # Don't change this! Return what you see printed out.
    print(np.sum(get_distance(X,np.array(range(n)))))

160239119987.1912
```

10 iterations after running k-means on only the First 25 datapoints, up to 3 digits after the decimal. Plot also the clustering result.

```
[8] # Run 10 iterations
    class_membership = kmeans(X[:25], 10)
    plot_class_membership(class_membership, 10)
    print('Purity:', overall_purity(class_membership))

Succefully, cluster complete!
Purity: 0.68
```

Implement PCA, and reduce the data dimension to d = 10, 100, and 500.

```
time_begin = time.time()
X = X - np.outer(np.ones(X.shape[0]), np.mean(X, axis=0))
U, Sigma, Vh = np.linalg.svd(X, full_matrices=False, compute_uv=True)

for d in [10, 100, 500]:
    print('Reduce the dimension =', d)
    X_SVD = np.dot(U[:, :d], np.diag(Sigma[:d]))
    class_membership = kmeans(X_SVD, 10)
    purity = overall_purity(class_membership)
    print('Finally Purity:', purity)

time_end = time.time()
print("time cost:", time_end-time_begin)

Reduce the dimension = 10
Succefully, cluster complete!
Finally Purity: 0.804
Reduce the dimension = 100
Succefully, cluster complete!
Finally Purity: 0.75
Reduce the dimension = 500
Succefully, cluster complete!
Finally Purity: 0.739
time cost: 0.674248456954956
```

Use random hashing (as promoted by the JL lemma) and reduce the feature dimension

```
time_begin = time.time()
for d in [10, 100, 500]:
    print('Result the dimension =', d)
    A = np.random.normal(0, 1, size=(d, X.shape[1]))
    X_jL = (1 / np.sqrt(d)) * A.dot(X.T).T
    class_membership = kmeans(X_jL, 10)
    purity = overall_purity(class_membership)
    print('Finally Purity:', purity)

time_end = time.time()
print("time cost:", time_end-time_begin)

Result the dimension = 10
Succefully, cluster complete!
Finally Purity: 0.472
Result the dimension = 100
Succefully, cluster complete!
Finally Purity: 0.727
Result the dimension = 500
Succefully, cluster complete!
Finally Purity: 0.75
time cost: 0.2876701354980469
```

Isomap, LLE

```
from scipy.sparse.csgraph import shortest_path

def isomap(X, threshold, dimension):
    dist = np.zeros((X.shape[0], X.shape[0]))
    for i in range(X.shape[0]):
        dist[i] = get_distance(X, X[i, :])
    adj = np.zeros((X.shape[0], X.shape[0])) + np.inf
    adj[dist<threshold] = dist[dist<threshold]
    dist_matrix = shortest_path(csgraph=adj)
    h = np.eye(X.shape[0]) - (1/X.shape[0]) * np.ones((X.shape[0], X.shape[0]))
    c = -1/(2*X.shape[0]) * h.dot(dist_matrix).dot(h)
    evals, evecs = np.linalg.eig(c)
    idx = evals.argsort()[::-1]
    evals, evecs = evals[idx][:dimension], evecs[:, idx][:, :dimension]
    z = evecs.dot(np.diag(evals**(-1/2)))

    return z.real
```

```
time_begin = time.time()
for d in [10, 100, 500]:
    print('Result the dimension =', d)
    X_isomap = isomap(X, 1000, d)
    class_membership = kmeans(X_isomap, 10)
    purity = overall_purity(class_membership)
    print('Finally Purity:', purity)

time_end = time.time()
print("time cost:", time_end-time_begin)
```

```
Result the dimension = 10
Succefully, cluster complete!
Finally Purity: 0.604
Result the dimension = 100
Succefully, cluster complete!
Finally Purity: 0.302
Result the dimension = 500
Succefully, cluster complete!
Finally Purity: 0.243
time cost: 47.24965167045593
```

Use sklearn's t-SNE to to reduce dimension to 1,2,3

```
from sklearn.manifold import TSNE

time_begin = time.time()

for d in [1, 2, 3]:
    print('Result the dimension =', d)
    X_embedded = TSNE(n_components=d, learning_rate='auto', init='random').fit_transform(X)
    class_membership = kmeans(X_embedded, 10)
    purity = overall_purity(class_membership)
    print('Finally Purity:', purity)

time_end = time.time()
print("time cost:", time_end-time_begin)
```

```
Result the dimension = 1
Succefully, cluster complete!
Finally Purity: 0.907
Result the dimension = 2
Succefully, cluster complete!
Finally Purity: 0.893
Result the dimension = 3
Succefully, cluster complete!
Finally Purity: 0.846
time cost: 34.38864779472351
```

By comparing the time cost, we can find that the time cost in 'random hashing' is the best, but the average final purity is the worst. By comparing the final purity, using sklearn's t-SNE can return the best final purity (average over than 0.85). but the time cost is too long (over that 30 seconds). So we may spend time efficiency on getting better purity and clustering result.

3. Hidden Markov Model spellchecker

- 4th word probabilities:

```
[34] ## FILL ME IN ##

     #WORD FREQUENCY
     #create an array of length V where V[k] returns the normalized frequency of word k in the entire data corpus. Do so by filling in this function.
     def get_word_prob(corpus):
         vocab = np.unique(corpus)
         length = len(vocab)
         word_prob = np.zeros(length)
         frequency = Counter(corpus)
         for i,key in enumerate(vocab):
             word_prob[i] =  frequency[key]/(len(corpus) + 0.)

         return word_prob

     word_prob =  get_word_prob(data['corpus'])

     #report the answer of the following:
     print ('prob. of "alice"', word_prob[vocab_hash['alice']])
     print ('prob. of "queen"', word_prob[vocab_hash['queen']])
     print ('prob. of "chapter"', word_prob[vocab_hash['chapter']])
```

```
prob. of "alice" 0.014548615047424706
prob. of "queen" 0.002569625514869818
prob. of "chapter" 0.0009069266523069947
```

same as the given result

- no smoothing

No Smoothing

```
[35] # Pr(word | prev word)
     # Using the uncorrupted corpus, accumulate the conditional transition probabilities. Do so via this formula:
     # pr(word | prev) = max(# times 'prev' preceded 'word' , 1) / # times prev appears
     # where again, we ensure that this number is never 0 with some small smoothing.
     def get_transition_matrix(corpus):
         SMALLNUM = 0.000001
         vocab = np.unique(corpus)
         length = len(corpus)
         # transition_matrix = np.ones((len(vocab),len(vocab)))*SMALLNUM
         transition_matrix = np.zeros((len(vocab),len(vocab)))
         for key in range(length-1):
             i = vocab_hash[corpus[key]]
             j = vocab_hash[corpus[key+1]]
             transition_matrix[j, i] = transition_matrix[j, i] + 1  # note key+1 is the original word

         for i in range(len(vocab)):
             transition_matrix[:, i] = transition_matrix[:, i] / corpus.count(vocab[i])

         return transition_matrix

     transition_matrix = get_transition_matrix(data['corpus'])
     print ('prob. of "the alice"', transition_matrix[vocab_hash['alice'],vocab_hash['the']])
     print ('prob. of "the queen"', transition_matrix[vocab_hash['queen'],vocab_hash['the']])
     print ('prob. of "the chapter"', transition_matrix[vocab_hash['chapter'],vocab_hash['the']])
     print ('prob. of "the hatter""', transition_matrix[vocab_hash['hatter'],vocab_hash['the']])
```

```
prob. of "the alice" 0.0
prob. of "the queen" 0.03968253968253968
prob. of "the chapter" 0.0
prob. of "the hatter"" 0.031135531135531136
```

same as the given result

- first 10 word closet to Alice:

```
[43] # The prior probabilities are just the word frequencies
     prior = word_prob  + 0.

     # Write a function that returns the emission probability of a potentially misspelled word, by comparing its probabilities against every word in the correct vocabulary

     def get_emission(mword):
         return np.zeros(V)

     def get_prob():
         prob = np.zeros(V)
         for i in range(V):
             prob[i] = prob_correct(mword, vocab[i])
         return prob

     #find the 10 closest words to 'abice' and report them
     idx = np.argsort(get_emission('abice'))[::-1]
     print([vocab[j] for j in idx[:10]])
```

```
['abide', 'alice', 'above', 'voice', 'alive', 'twice', 'thick', 'dance', 'stick', 'prize']
```

same as the given result.

· Recovery rate and word fixing

```
alices alices
Correct world:alices
adventures adventures
Correct world:adventures
in in
Correct world:in
wonderland wonderland
Correct world:wonderland
by yb
Correct world:by
lewis lewia
Correct world:lewis
carroll carroll
Correct world:carroll
the the
Correct world:the
millennium millennium
Correct world:millennium
fulcrum fulcrkm
Correct world:fulcrum
edition edition
Correct world:edition
30 30
Correct world:30
contents contents
Correct world:contents
chapter chapter
Correct world:chapter
```

corrupt word: "yb"
correct word: "by"

```
to to
Correct world:to
get get
Correct world:get
very very
Correct world:very
tired tired
Correct world:tired
of of
Correct world:of
sitting sitting
Correct world:sitting
by by
Correct world:by
her her
Correct world:her
sister sister
Correct world:sister
0.759 0.907
```

By checking the output, we have most correct spelling words, like 'the', 'of', 'her', etc.
We also have some non-correct spelling words, like 'lewia' (should be 'lewis'), 'yb' (should be 'by'), etc.
The recovery rate of fixed corpus is 0.907.