عنوان:

حل مسئله Part Of Speech Tagging

به کمک Hidden Marcov Models

نویسنده: ادیب مردوخی شماره دانشجویی : ۹۵۱۷۰۲۳۱۴۷ ادرس ایمیل: Adibmrunn@gmail.com

استاد راهنمای پروژه: دکتر فردین اخلاقیان

چکیده:

مدل های پنهان مارکوو (HMM) مدل های آماری هستند که به خاطر کاربردشان در یادگیری تقویتی (reinforcement learning) و شناسایی الگوهای زمانی (temporal pattern recognition) شناخته میشوند و در چند دهه اخیر مقالات فراوانی راجع به آنها به چاپ رسیده است. در این پروژه کاربردی از HMM در حل مسئله برچسب‌زدن قسمت‌های متن (part of speech tagging or POS) را نشان میدهیم. مسئله POS در دهه 1960 بیان شد و در ابتدا دقت الگوریتم های آن در نزدیکی 70٪ بود تا اینکه گروهی از دانشمندان اروپایی در دهه 1980 با استفاده از HMM ها دقت آن را به نزدیکی 93٪ رساندند و امروزه دقت الگوریتم های حل این مسئله در نزدیکی 97٪ قرار دارد. مسئله POS قسمت جدایی‌ناپذیر از پردازش زبان طبیعی (natural language processing) است زیرا در مواردی یافتن چه قسمتی بودن یک واژه بدون فهمیدن معنا (Semantics) ممکن نیست.

در این پروژه ابتدا خود HMM ها و الگوریتم های مربوط به آن (Forward, Backward, Viterbi, Baum-welch) پیاده سازی شده است و سپس یک بار با استفاده از کتابخانه‌ی آمده pomegranate و یک بار با توابع دست‌نویس مسئله POS را حل کرده‌ایم و به دقتی در نزدیکی 96٪ رسیده ایم که نتیجه مطلوبی است که قدرت HMM ها را به خوبی نشان میدهد.

مقدمه:

طبقه بندی خودکار متون از موارد کاربرد الگوریتم های یادگیری ماشینی در مبحث بازیابی اطلاعات میباشد که در حوزه پردازش زبان‌های طبیعی از جمله تحلیل‌های پراهمیت می‌باشد.

در طول 20 سال گذشته اسناد متنی دیجیتال به صورت تصاعدی رشد پیدا کرده-اند. یکی از نتایج این رشد تصاعدی اهمیت پیدا کردن دسته -بندی اسناد بر اساس محتوا می‌باشد. دسته‌بندی اسناد را می‌توان به عنوان یکی از تکنیک‌های متن‌کاوی در هوش مصنوعی دانست. متن کاوی، خود زیر مجموعه‌ای از داده‌کاوی است که تمرکز آن بر استخراج داده‌های مفید و کشف دانش از داده‌های متنی می‌باشد.

اما یک کامپیوتر نمی‌تواند زبانِ طبیعیِ محاوره‌ایِ ما را متوجه شود. برای همین نیاز است تا یک مجموعه عملیات بر روی این زبان طبیعی انجام شود تا بتوان آن را برای کامپیوتر قابل فهم کرد. بعد از آن می‌توان از قدرتِ کامپیوتر در محاسبات و یادگیری ماشین استفاده کنیم تا اطلاعاتِ ارزشمندی را از میان این داده‌ها استخراج کنیم.

متن کاوي يكي از زمينه هاي است كه به دنبال استخراج اطلاعات مفيد، از داده هاي متني بدون ساختار، به وسيله شناسايي و اكتشاف الگوها مي باشد. ايده اصلي متن كاوي، يافتن قطعات كوچك اطلاعات ازحجم زياد داده هاي متني، بدون نياز به خواندن تمام آن است. متن کاوی اطلاعات متنی غیرساخت یافته را استفاده می کند و آن را برای کشف ساختار و معناهای ضمنی پنهان در متن بررسی می کند.

دسته بندی متن (Part of speech tagging or POS) یکی از روش های آگاهانه ی یادگیری ماشین است که از آن برای برچسب زنی متن ها در یکی از دسته های مشخص استفاده می شود. در واقع دسته بندی به عمل جمع آوری اسناد متنی و پردازش آن ها برای کشف دسته ی مناسب شان گفته می شود.

نظرکاوی، شناسایی زبان، کمک به موتورهای جستوجو، تحلیل نظرات کاربران شبکه‌های اجتماعی، فیلتر متن‌های اسپم و توصیف متن از کاربردهای دسته بندی متن هستند. در روش های مبتنی بر یادگیری ماشین یک طبقه بند متن، از روی مجموعه داده های برچسب گذاری شده آموزش می بیند. طبقه بندی احساسی مبتنی بر این روش می تواند به عنوان یک مسئله آموزش نظارت شده (supervised) درنظر گرفته شود.

معرفی زمینه کار:

در مسئله POS نقش هر واژه در جمله را (مثل اسم، فعل، حرف اضافه یا ...) در یک جمله مستقل از پاراگراف مشخص میکنیم. این مسئله از انجایی شروع به پیچیده شدن میکند که معنی بعضی واژه‌ها مبهم است و میتواند معنی کل جمله را تغییر دهد و فقط با توجه به قبل و بعد آن واژه قابل تشخیص میشود. لازم به ذکر است که در بعضی موارد فقط با داشتن دانش معنایی از جمله، واژه‌ها قابل برچسب گذاری میشوند.

روش‌های متفاوتی برای حل این مسئله وجود دارد که به طور کلی در ۳ دسته قرار میگیرند. دسته اول روش های آماری، دسته دوم روش های مبتنی بر گرامر و قواعد زبان (rule based) و دسته سوم روش‌های ترکیبی (hybrid) هستند که از دو دسته قبلی که برای تعیین نقش واژه به همدیگر کمک میکنند.

در روش rule-based از regular expression ها و قواعد گرامری استفاده میشود به همین دلیل مقیاس‌پذیر (Scalable) نیست. در این روش ابتدا یک لیست از معانی ممکن برای واژه از لغتنامه (دیکشنری) استخراج میشود و سپس توسط قواعد زبانی که در پایگاه وارد شده است تمام حالات ممکن برای طبقه بندی کل جمله امتحان میشود و یک یا چند حالت محتمل‌تر را به عنوان متن تشخیص داده شده برمیگرداند.

در روش های آماری از HMM ها یا SVM یا مدل های مبنی بر گراف یا مدل های مبتنی بر شبکه عصبی استفاده شده است و همه آنها نتایج قابل قبولی گرفته شده است. روش‌های آماری محدودیت‌های هم دارند زیرا بسیار به کیفیت و حجم پایگاه جملات ما بستگی دارد. یکی از محدودیت‌های روش‌های آماری واژه‌های ناشناخته است که یعنی برای تشخیص نقش آن باید حدس بزنیم و حدس‌های بیشتر به معنی خطاهای بیشتر است. یکی دیگر از مشکلات روش‌های آماری سخت بودن ردیابی (traceability) این است که ماشین چه چیزی را اشتباه یادگرفته و تصحیح این اشتباهات است. مشکلات مربوط به پایگاه جملات زیاد و بسیار جدی هستند ولی مشاهده میشود که دغدغه بیشتر مقالات روش‌های بهینه سازی شده است تا بهبود کیفیت و حجم پایگاه داده.

در این پروژه قصد داریم از مدل مارکوو پنهان (HMM) ها استفاده کنیم. مدل مارکوو یک مدل آماری است برای مدل کردن سیستم‌های که به صورت تصادفی تغییر حالت میدهند. در این مدل زمان را به صورت برش‌ها یا snapshot هایی میبینم که در هرکدام یک متغییر حالت (state) و یک مشاهده (observation) داریم. در مدل مارکوو فرض میکنیم که در هر برش زمانی مجموعه حالت‌ها و مجموعه مشاهده‌ها ثابت است و چیزی از آن کم یا زیاد نمیشود. فرض دیگر مدل مارکوو (Marcov Assumption) این است که حالت فعلی فقط به تعداد محدودی از حالت‌های قبلی وابسته است. مدل مارکوو پنهان (HMM) حالت کلی‌تر از مدل مارکوو است که در آن فقط مشاهده‌ها را دارم و میدانیم هر حالت یک خروجی دارد و بر اساس خروجی دنباله حالت‌ها را میابیم و در این فرایند فقط از قضیه احتمال شرطی و قضیه بیز استفاده میشود.

در طول حل این مسئله به سه سوال کلیدی برمیخوریم:

۱. احتمال اینکه یک مدل این دنباله مشاهده‌ها را به وجود اورده باشد چقدر است؟ برای جواب دادن این سوال به الگوریتم forward-backward نیاز داریم.
۲. بهترین دنباله حالت ممکن که این دنباله مشاهده‌ها را توجیه کند کدام است؟ برای جواب دادن این سوال به الگوریتم Viterbi نیاز داریم.
۳. با داشتن مجموعه‌ای از مشاهده‌ها چگونه مدلی که آنها را به وجود آورده است یاد بگیریم؟ برای جواب دادن این سوال به الگوریتم Baum-Welch نیاز داریم.

متغیرهای که در مدل مارکو پنهان استفاده میشود در زیر اورده شده است.

$HMM = \{N, M, A, B, \pi\}$ $O = \{o_1, o_2, \dots, o_T\}$

N: State space

M: Observation space or Language of observations

A: Transition Probability (probability of going from state s_i to s_j)

B: Emission Probability (probability of seeing observation o_j in state s_i)

π: Beginning Probability (probability of starting from state s_i)

O: Observation at time t is saved in variable o_t

کارهای مشابه قبلی:

در این قسمت تعدادی از مقالات مرتبط را بیان میکنیم.

در مقاله [Stratos 2016] مسئله POS tagging به صورت غیرنظارت شده (unsupervised) با دقت 74% و با استفاده از گونه خاصی از HMMها حل شده است. در مقاله [Lee 2000] همین مسئله با HMM ها با تغییر Markov assumption از استقلال شرطی به mutual independence نتایج بهتری به دست امد. در مقاله [brill 1994] از rule-based tagging برای واژه های ناشناخته استفاده شده و مفهوم جدید به اسم k-best tagger بیان شده که چندین برچسب را در زمان عدم اطمینان میتوان به یک واژه نسبت داد که این دو در کنار هم دقت را تا 99٪ افزایش داده‌اند. در مقالاتی مثل [Thede & Harper 1999] تاثیر مرتبه‌های بالاتر از مدل مارکو (یعنی به جای نگاه به یک واژه قبلی به دو یا سه واژه قبلی نگاه کنیم) مطالعه شده است که نتیجه این شد که فرایند اموزش بسیار بیشتر طول خواهد کشید و در عوض تنها 0.5٪ بهبود در عملکرد دیده شد که چندان معامله خوبی نیست. در مقالات دیگری هم تاثیر روش‌های مختلف smoothing در نظر گرفته شده است زیرا ماتریس‌های HMM ماتریس‌های بسیار خلوت (sparse) بوده و این روش با کاهش تعداد صفرها باعث میشود مدل احتمالات بیشتری را در نظر بگیرد.

ایده اصلی پروژه:

در این پروژه قصد داشته‌ایم قدرت HMM ها را نشان دهیم که چگونه مسئله‌ای به سختی POS tagging را با پیچیدگی زمانی خطی با دقتی قابل قبول حل کرد. در زیر خواص هرکدام از الگوریتم ها را آورده ایم.

الگوریتم Forward-Backward (برای محاسبه تمام مسیرهایی که میتواند دنباله مشاهدات ورودی را تولید کند) پایه اصلی بسیاری از application های است که با دنباله از مشاهدات noisy سروکار دارند. این الگوریتم را میشود با استفاده از dynamic programming با پیچیدگی زمانی خطی و پیچیدگی حافظه O(|f|*t) پیاده سازی کرد که f در اینجا forward message میباشد (البته حالت دیگری نیز وجود دارد که میشود آن را با پیچیدگی حافظه O(|f|*Log(t)) نیز پیاده سازی کرد به شرط صرف زمان بیشتر). تنها مشکلی که از این الگوریتم باقی میماند این است که نمیتوان به صورت Online استفاده کرد و مشاهدات جدید به دنباله مشاهدات اضافه کرد که این مشکل نیز با استفاده از Fix-lag smoothing قابل حل است.

الگوریتم Viterbi (برای محاسبه محتمل‌ترین مسیری که میتواند دنباله مشاهدات ورودی را تولید کند) برپایهdynamic programming با زمان و حافظه خطی است که برای مدل کردن وابستگی‌های دوربرد بین داده مناسب است.

با توجه به موارد بالا اگر بخواهیم HMM را با سایر روش‌های حل بررسی کنیم به این نتیجه میرسیم که دقت HMM از همه بجز مدل‌های مبتنی بر شبکه عصبی عملکرد بهتری دارد و اختلاف‌کوچک خود را با شبکه‌های عصبی از طریق سرعت بیشتر هم در عملکرد و هم در یادگیری، سادگی توابع، ماتریسی شدن تمام مراحل محاسبات بدون نیاز به مشتق‌گیری و معکوس گرفتن از ماتریس‌ها و تعداد کم Hyper Parameters جبران کرده است.

توضیحات انجام پروژه و پیاده سازی:

قسمت اول:

در ابتدا قبل از حل مسئله POS به خود HMM پرداخته ایم و تمامی کدها به دلیل قابل مشاهده بودن و شفافیت نتایج در Jupyter notebook هایی نوشته شده است. در فایل اول الگوریتم‌های Forward-Backward و Viterbi و Baum-Welch پیاده سازی شده است. (نکته: در ابتدای پروژه به این دلیل که داده ای در دست نبود و قسمت دوم پروژه تعریف نشده بود این قسمت بر محور یک آرایه با مقادیر راندوم تعریف شده است.) در زیر متغیرهایی که در طول فایل بسیار تکرار شده است را آورده‌ایم.

```
N=3 # size of states space
M=4 # size of oservations space
T=5 # number of times we see an observation
```

```
# generate a random sequence of observations
# numpy.random.randint(low, high=None, size=None, dtype=int)
O = np.random.randint(0,M,T)
print(O)
```

```
[0 3 3 3 0]
```

N مجموعه حالت‌ها (State) موجود در HMM است.

M مجموعه مشاهدات (Observation) موجود در HMM است.

T اندازه دنباله مشاهداتی است که ما در دسترس داریم.

O دنباله مشاهدات ما هستند.

```
# observation matrix
b = np.random.random((N,M))
b /= np.array([b.sum(axis=0)])
print(b)
```

```
[[0.29422203 0.07484759 0.03951524 0.13112228]
 [0.44855994 0.39854757 0.34780568 0.39427893]
 [0.25721804 0.52660484 0.61267908 0.47459878]]
```

```
# probabilites for individual events have to sum to 1
print(b.sum(axis=0))
```

```
[1. 1. 1. 1.]
```

b ماتریس مشاهدات ما است. هر سطر نماینده یک حالت است. و هر ستون نماینده یک عضو از مجموعه مشاهدات است.

b_{11} میگوید وقتی در حالت S_1 هستیم با چه احتمالی O_1 را مشاهده میکنیم.

در اینجا این ماتریس را اعداد تصادفی تشکیل داده‌اند که بعدا مقادیر درست‌تر توسط الگوریتم Baum-welchبه دست می‌آید.

در یکی از مقالات به اهمیت نرمال‌سازی ماتریس‌ها اشاره شده است پس ماتریس را نرمال میکنیم. برای اطمینان از صحت درست نرمال شدن اعداد هر ستون را جمع میکنیم و حاصل باید یک شود زیرا مجموع حالت‌های که میشود با مشاهده O فرض کرد با توجه به استقلال مشاهدات باید یک شود.

```
#state transition matrix
a = np.random.random((N,N))
a /= np.array([a.sum(axis=-1)]).T
print(a)
print(a.sum(axis=1))
```

```
[[0.24516257 0.42538428 0.32945316]
 [0.1551306  0.49998497 0.34488443]
 [0.31759074 0.28453531 0.39787395]]
[1. 1. 1.]
```

a ماتریس انتقال است. هر سطر و هر ستون نماینده یک حالت هستند.

a_{11} میگوید که با چه احتمالی میتوان از حالت S_1 باز به حالت S_1 رفت.

در اینجا این ماتریس را اعداد تصادفی تشکیل داده‌اند که بعدا مقادیر درست‌تر توسط الگوریتم Baum-welch به دست می‌آید.

باز در اینجا هم نرمال‌سازی انجام میدهیم و برای صحت از درستی اعداد هر سطر را جمع میکنیم که حاصل آن باید یک شود.

```
# generate the priors
pi = np.random.random(N)
pi /= pi.sum()
print(pi)
```

```
[0.56694725 0.27998804 0.15306471]
```

ماتریس پی به ما میگوید که در شروع دنباله مشاهدات با چه احتمالی در کدام حالت هستیم. در اینجا نیز نرمال‌سازی انجام شده است.

```
hmm =(O,pi,a,b,N,M,T)
```

در اینجا هم کل اطلاعات لازم در یک چندتایی (tuple) ذخیره کرده ایم.

```
def forward(O,pi,a,b,N,M,T):
    fwd = np.zeros((T,N))

    #initialization
    fwd[0] = pi*b[:,O[0]]

    #induction:
    for t in range(T-1):
        fwd[t+1] = np.dot(fwd[t],a)*b[:,O[t+1]]

    return fwd

print(forward(*hmm))
```

```
[[0.16680837 0.12559142 0.03937101]
 [0.00955647 0.05715228 0.0540733 ]
 [0.00372153 0.01893572 0.0210597 ]
 [0.0013818  0.00671965 0.00765804]
 [0.00112196 0.0027481  0.00149693]]
```

در زیر تعریف ریاضی الگوریتم آورده شده است. آلفا همان Forward است.

$$\lambda = (A, B, \pi) \qquad \alpha_t(i) = P(O_1\, O_2 \cdots O_t, q_t = S_i|\lambda)$$

پس الگوریتم Forward احتمال دیدن دنباله مشاهدات $O_1, ...,O_t$ و اینکه در زمان t در حالت S_i باشیم به شرط اینکه مدل داده شده باشد را برای ما حساب میکند. یعنی اگر در زمان t در حالت S_i باشیم احتمال اینکه قبلا در چه حالت‌های بوده ایم را حساب میکند.

هر سطر در این ماتریس جلوتر میرویم اعداد کوچکتر میشوند که منطقی است. چون سطر بعد با توجه به احتمال انتقال به حالتی دیگر با در نظر گرفتن احتمال مشاهدات است که هر دو اعدادی کوچکتر از یک هستند.

```
# sum of all the forward probabilities at the last time step
def full_prob(fwd):
    return fwd[-1].sum()

print(full_prob(forward(*hmm)))
```

```
0.005366989247259792
```

در اینجا تابع full_prob را هم اضافه کرده ایم که هم در نرمال کردن و هم در تعیین قدرت مدل به کار ما می‌آید.

```
def backward(O,pi,a,b,N,M,T):
    bk=np.zeros((T,N))

    #initialization:
    bk[T-1] = 1

    #induction:
    for t in reversed(range(T-1)):
        bk[t] = np.dot(bk[t+1]*b[:,O[t+1]],a.T)
    return bk

print(backward(*hmm))
```

```
[[0.01583352 0.01700768 0.01498074]
 [0.04394399 0.04719749 0.04160271]
 [0.12189395 0.1307065  0.11578202]
 [0.34768387 0.35862656 0.32341369]
 [1.         1.         1.        ]]
```

الگوریتم Backward احتمال دیدن دنباله مشاهدات $O_t, ...,O_T$ به شرطی که در زمان t در حالت Si باشیم و مدل داده شده باشد را برای ما حساب میکند. یعنی اگر در زمان t در حالت S_i باشیم احتمال اینکه بعدا به چه حالتی میرویم را حساب میکند. تعریف ریاضی آن در زیر اورده شده است.

$$\beta_t(i) = P(O_{t+1}\, O_{t+2} \cdots O_T|q_t = S_i, \lambda)$$

سطر اخر ماتریس یک است چون اینکه الان در چه حالتی هستم داده شده است. هر سطر که در ماتریس عقب میرویم اعداد کوچکتر میشود چون در حال گمان زدن آینده هستیم و هرچقدر زمان دورتری را بخواهیم پیشبینی کنیم احتمال اشتباه زیادتر است ودر نتیجه احتمال گمان درست کمتر.

```
def gamma(fwd,bk,fp):
    return (fwd*bk)/fp

print(gamma(forward(*hmm),backward(*hmm),full_prob(forward(*hmm))))
```

```
[[0.49211274 0.39799197 0.10989528]
 [0.07824673 0.50259912 0.41915415]
 [0.08452263 0.46115656 0.45432082]
 [0.08951567 0.44901254 0.46147179]
 [0.20904832 0.51203824 0.27891344]]
```

وقتی Forward-Backward کامل شد میتوانیم از انها را ترکیب کنیم و احتمال اینکه بودن در هر حالت را در هر لحظه از زمان یافت. در نهایت آن را نرمال هم میکنیم.

```
# d(t, i): path with the highest probability of seeing the first t observation then ending in State i
# ph(t, i): if we are in state i in time t, what is the privious state?
def viterbi(O,pi,a,b,N,M,T):
    d = np.zeros((T,N))
    ph = np.zeros((T,N), dtype=np.int)

    #initialization
    d[0] = pi*b[:,O[0]]
    ph[0] = 0

    #recursion
    for t in range(1,T):
        m = d[t-1] * a.T
        ph[t] = m.argmax(axis=1)
        d[t] = m[np.arange(N), ph[t]] * b[:,O[t]]

    #termination
    Q = np.zeros(T,dtype=np.int)
    Q[T-1] = np.argmax(d[T-1])
    Pv = d[T-1, Q[T-1]]

    #path back-tracking
    for t in reversed(range(T-1)):
        Q[t] = ph[t+1, Q[t+1]]

    return Q

print(viterbi(*hmm))
```

```
[0 1 1 1 1]
```

$$\delta_t(i) = \max_{q_1, q_2, \cdots, q_{t-1}} P[q_1\, q_2 \cdots q_t = i,\, O_1\, O_2 \cdots O_t | \lambda]$$

$$\delta_{t+1}(j) = [\max_i \delta_t(i) a_{ij}] \cdot b_j(O_{t+1}).$$

الگوریتم Viterbi مسیری که بیشترین احتمال را دارد با توجه به مشاهدات به ما میدهد. در اینجا از دو ماتریس کمکی d و ph هم استفاده شده است. فرق این با Forward-Backward این است که در Viterbi مسیری که از آن امده‌ایم را ذخیره میکنیم که در نهایت بتوانیم از ان استفاده کنیم.

d ماتریس کل مسیرها است و d(t, i) مسیری است که به حالت i ختم میشود و با توجه به مشاهدات بیشترین احتمال رخ دادن را دارد.

ph(t, i) با توجه به زمان و حالت فعلی حالت قبلی را به ما میدهد. یعنی اعداد داخل آن شماره حالات هستند.

m ارایه‌ای به اندازه فضای حالت‌مان است که با توجه به مسیرمان و ماتریس انتقال، احتمال رفتن به حالات بعدی را حساب میکند. و سپس حالتی که بیشتری احتمال را دارد که به آن سفر کنیم را در ph[t, j] میریزیم و سپس با ضرب کردن در احتمال دیدن هرکدام از مشاهدات بعدی را حساب میکنیم و در d[t, j] میریزیم.

عکس زیر حالت گویاتری از الگوریتم است که از Vectorize نشده است. (این کد در فایل پروژه وجود ندارد.)

```
def viterbi(O,pi,a,b,N,M,T):
    d=np.zeros((T,N))
    ph=np.zeros((T,N),dtype=np.int)

    #initialization
    for i in range(N):
        d[0,i]=pi[i]*b[i,O[0]]
        ph[0,i]=0

    #recursion
    for t in range(1,T):
        for j in range(N):
            m=np.zeros(N)
            for i in range(N):
                m[i]=d[t-1,i]*a[i,j]
            ph[t,j]=m.argmax()
            d[t,j]=m.max()*b[j,O[t]]

    #termination
    m=np.zeros(N)
    for i in range(N):
        m[i]=d[T-1,i]
    Pv=m.max()

    #path back-tracking
    Q=np.zeros(T,dtype=np.int)
    Q[T-1]=m.argmax()
    for t in reversed(range(T-1)):
        Q[t]=ph[t+1,Q[t+1]]

    return Q

print viterbi(*hmm)
```

متغیر Pv در اینجا بی کاربرد است چون احتمال رفتن به حالت بعدی را برای زمانی خارج از بازه زمانی و دنباله مشاهدات تخمین میزند.

متغیر Q نیز مسیر نهایی را از ماتریس ph بیرون میکشد. توجه شود که حالات را از اخرین زمان به اولین زمان بیرون میکشیم زیرا اخرین زمان است که مسیر را مشخص میکند نه زمان اول پس بزرگترین احتمال را از سطر اخر ph حالت ما را مشخص میکند و سپس همینطور به سمت سطر اول حرکت کرد و برزگترین احتمال حالت هر زمان را مشخص میکند.

Argmax تابعی است که شماره خانه ای که در آن بزرگترین عدد ارایه قرار دارد را برمیگرداند.

(اگر هرکدام از توضیحات کافی نبود میتوانید با چاپ کردن ان متغیر محتوای داخل آن را ببینید تا درک بهتری از کارکرد الگوریتم پیدا کنید.)

```
# The transition probability computed for each timestep
def xi(fwd,bk,fp,O,pi,a,b,N,M,T):
    return  fwd[:-1].reshape((T-1,N,1))*\
            a.reshape((1,N,N))*\
            b[:,O[1:]].T.reshape((T-1,1,N))*\
            bk[1:].reshape((T-1,1,N))/fp

print(xi(forward(*hmm),backward(*hmm),full_prob(forward(*hmm)),*hmm))
```

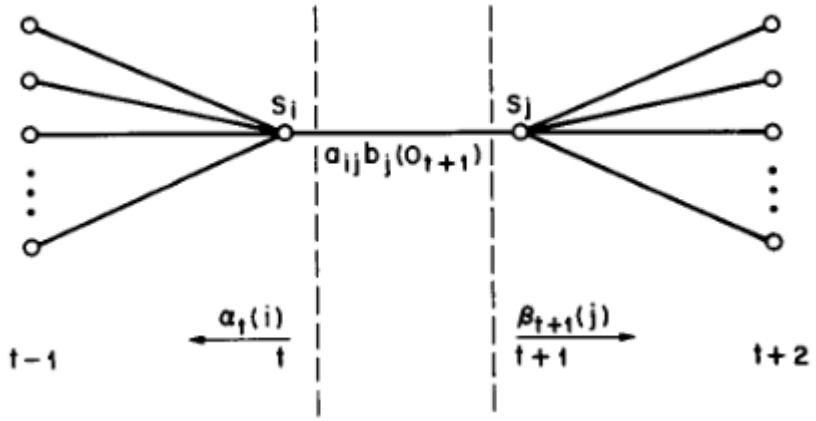
الگوریتم Baum-Welch.

```
def xi(fwd,bk,fp,O,pi,a,b,N,M,T):
    ret=np.zeros((T-1,N,N))
    for t in range(T-1):
        for i in range(N):
            for j in range(N):
                ret[t,i,j]=(fwd[t,i]*a[i,j]*b[j,O[t+1]]*bk[t+1,j])/fp
    return ret

print xi(forward(*hmm),backward(*hmm),full_prob(forward(*hmm)),*hmm)
```

حالت غیر Vectorize شده همین الگوریتم که در کدها نیست.

$$\xi_t(i, j) = P(q_t = S_i, q_{t+1} = S_j | O, \lambda).$$

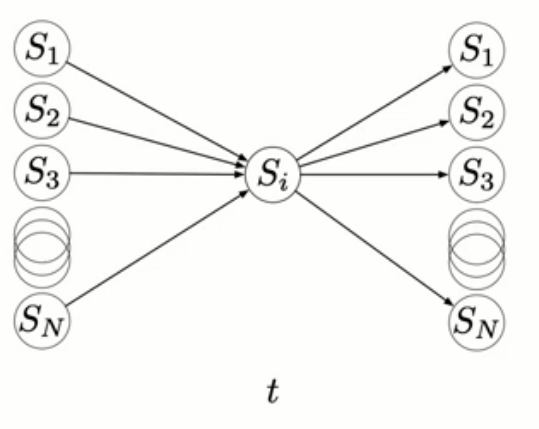


در اینجا فرمول ریاضی و شکلی از نحوه عملکرد الگوریتم داریم با این تفاوت که در الگوریتم این عبارت نرمال شده اشت.

اگر بخواهیم این عبارت را به مراحل قبل ربط دهیم به نتیجه‌ي شکل زیر میرسیم.

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i, j).$$

در اینجا gamma را هم ترسیم کرده‌ایم که شفاف‌تر شود.



حال برای توضیح ادامه الگوریتم به دو احتمال زیر نیاز داریم.

$$\sum_{t=1}^{T-1} \gamma_t(i) = \text{expected number of transitions from } S_i$$

$Gamma_t(i)$ احتمال بودن در حالت S_i در زمان t را بیان میکند. میتوان با جمع زدن این عبارت روی تمام زمان‌ها احتمال اینکه هیچوقت در S_i بوده باشیم را بدست بیاوریم.

$$\sum_{t=1}^{T-1} \xi_t(i, j) = \text{expected number of transitions from } S_i \text{ to } S_j.$$

$xi_t(i, j)$ احتمال رفتن از S_i به S_j در زمان t را بیان میکند. میتوان با جمع زدن رو تمام زمان‌ها احتمال اینکه هیچوقت از S_i به S_j رفته باشیم را بدست آوریم.

حال در قسمت بعدی الگوریتم مقادیر pi و a و b را به روز رسانی میکنیم.

```
# The expected prior value can be estimated by simply looking at the
# expected value (gamma) at the first point in time
def exp_pi(gamma):
    return gamma[0]

print(exp_pi(gamma(forward(*hmm),backward(*hmm),full_prob(forward(*hmm)))))
```

```
[0.49211274 0.39799197 0.10989528]
```

حال در این قسمت از الگوریتم Baum-welch عبارت Pi را به دست می‌آوریم که احتمال بودن در حالت خاصی در اولین زمان است که با توجه به تعریف Gamma برابر مقدار gamma در زمان صفر میشود.

$$\bar{a}_{ij} = \frac{\text{expected number of transitions from state } S_i \text{ to state } S_j}{\text{expected number of transitions from state } S_i}$$

$$= \frac{\sum_{t=1}^{T-1} \xi_t(i, j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

در بالا مراحل ریاضی به روز رسانی ماتریس انتقال را میبینیم. در زیر همین مرحله تبدیل به کد شده است.

```
# The expected transistions is the sum (for different timestep) of transition
# probabilities xi normalized by the corresponding emitting state probabilities
def exp_a(gamma,xi,N):
    return xi[:].sum(axis=0)/gamma[:-1].sum(axis=0).reshape(N,1)

fw = forward(*hmm)
bk = backward(*hmm)
fp = full_prob(fw)
g  = gamma(fw,bk,fp)
x  = xi(fw,bk,fp,*hmm)

print(exp_a(g,x,N))
```

```
[[0.1037133  0.50497286 0.39131383]
 [0.07147226 0.5645922  0.36393554]
 [0.17628936 0.36444448 0.45926616]]
```

خب حال ماتریس مشاهدات را به روز رسانی میکنیم. در زیر تعریف ریاضی و کد آن را میبینیم. (علامت bar در بالای a, b به معنی به روز رسانی شده این عبارت است.)

$$\bar{b}_j(k) = \frac{\text{expected number of times in state } j \text{ and observing symbol } v_k}{\text{expected number of times in state } j}$$

$$= \frac{\sum_{\substack{t=1 \\ \text{s.t. } O_t = v_k}}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}.$$

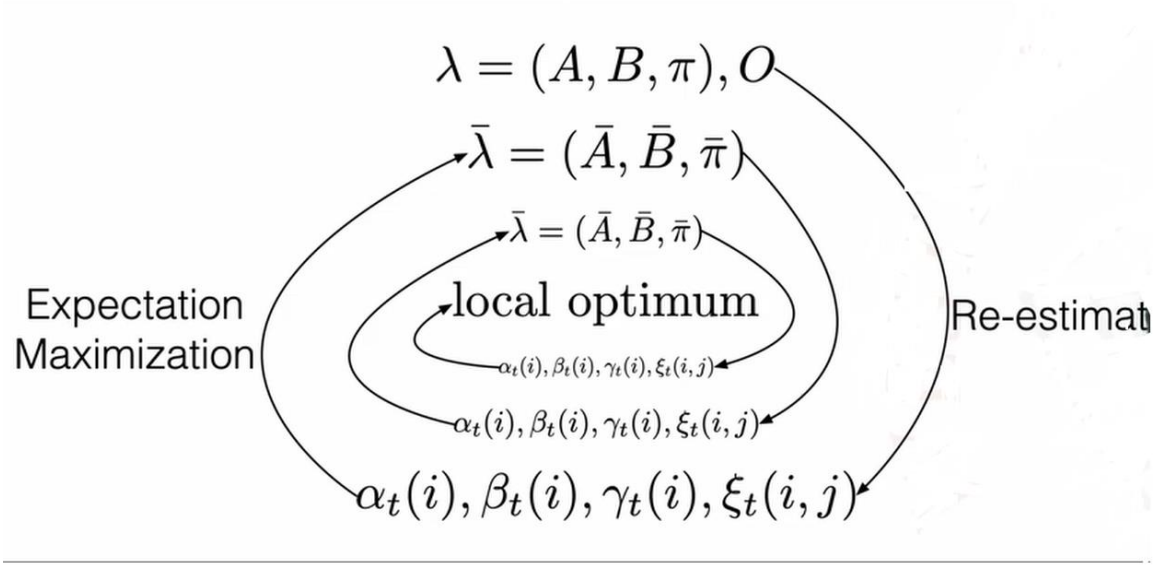
```
# the expected observation likelihood is sum of probabilities of given states at times where
# the specific observation occured normalized by the sum of all the probilities in time
# for the given state
def exp_b(gamma,O,N,M):
    return np.array(list(map(lambda k: np.sum(gamma[O==k],axis=0)/np.sum(gamma,axis=0), np.arange(M)))).T

fw = forward(*hmm)
bk = backward(*hmm)
fp = full_prob(fw)
g  = gamma(fw,bk,fp)

print(exp_b(g,O,N,M))
```

```
[[0.73539665 0.         0.         0.26460335]
 [0.39178183 0.         0.         0.60821817]
 [0.22555909 0.         0.         0.77444091]]
```

حال که تعریف‌ها تمام شد به سراغ خود عملیات به روز رسانی میرویم. همانطور که میدانیم که الگوریتم Baum-welch یک متد EM یا Expectation Maximization است که بهینه محلی (Local Optimal) را پیدا میکند (تابه حال هیچ روش شناخته شده‌ای برای یافتن بهینه جهانی (global optimal) یافت نشده است). الگوریتم Baum-welch به صورت تکراری(iterative) به بهینه محلی همگرا میشود که این روش را به اسم Gradient descent هم میشناسیم.



همانطور که در شکل میبینیم از مدل‌مان، متغییرهای‌مان را محاسبه میکنیم بعد روی EM میزنیم و سپس مدل جدیدمان را به دست می‌آوریم و این روش اثبات شده قطعا همگرا میشود. همین عکس را اگر تبدیل به کد کنیم شبیه عکس زیر خواهد شد.

```
print('Initial probability: {}'.format(full_prob(forward(*hmm))))

hmm_new=hmm
for i in range(15):
    fw = forward(*hmm_new)
    bk = backward(*hmm_new)
    fp = full_prob(fw)
    g  = gamma(fw,bk,fp)
    x  = xi(fw,bk,fp,*hmm_new)

    pi_new = exp_pi(g)
    a_new  = exp_a(g,x,N)
    b_new  = exp_b(g,O,N,M)

    err = np.concatenate(((pi_new-hmm_new[1]).ravel(),(a_new-hmm_new[2]).ravel(),(b_new-hmm_new[3]).ravel()))

    hmm_new = (O,pi_new,a_new,b_new,N,M,T)

    print('Update #{} probability: {} -- mean error: {}'.format(i+1,full_prob(forward(*hmm_new)),np.mean(err**2)))
```

```
Initial probability: 0.005366989247259792
Update #1 probability: 0.051064857052248995 -- mean error: 0.05771785410418955
Update #2 probability: 0.07094211999280131 -- mean error: 0.00396883133331852
Update #3 probability: 0.09603175883786529 -- mean error: 0.003350119865485326
Update #4 probability: 0.11277771061930612 -- mean error: 0.0012735524504434638
Update #5 probability: 0.1204017979465175 -- mean error: 0.0003245524207321524
Update #6 probability: 0.1246232203473026 -- mean error: 0.0001649316486886098
Update #7 probability: 0.1280278518201862 -- mean error: 0.00015484004081682884
Update #8 probability: 0.13143492544939248 -- mean error: 0.00017666660529634364
Update #9 probability: 0.13527159603211697 -- mean error: 0.0002279714869739399
Update #10 probability: 0.14019818530610198 -- mean error: 0.0003295356487009543
```

همانطور که در عکس میبینیم با هر تکرار الگوریتم (که اینجا عدد محدودی را قرار داده ایم تا اینکه حلقه پایان پذیرد ولی بهتر این است که با مقدار err آن را بسنجیم و اگر تغیرات err از مقدار مشخصی کوچتر بود حلقه تمام شود.) مقدار خطا کمتر و احتمال Full_prob بیشتر میشود.

خب در اینجا به پایان قسمت اول پروژه میرسیم. در این بخش سه الگوریتم مهم مربوط به HMM ها را بررسی کردیم.

قسمت دوم:

در این قسمت از پروژه به حل POS tagging میپردازیم یک بار با استفاده از کتابخانه‌های موجود و یک بار هم با استفاده از توابع ساده‌ای نوشته‌ایم تا شمایی کلی از آنچه داخل کتابخانه‌ها وجود دارد را بهتر درک کنیم.

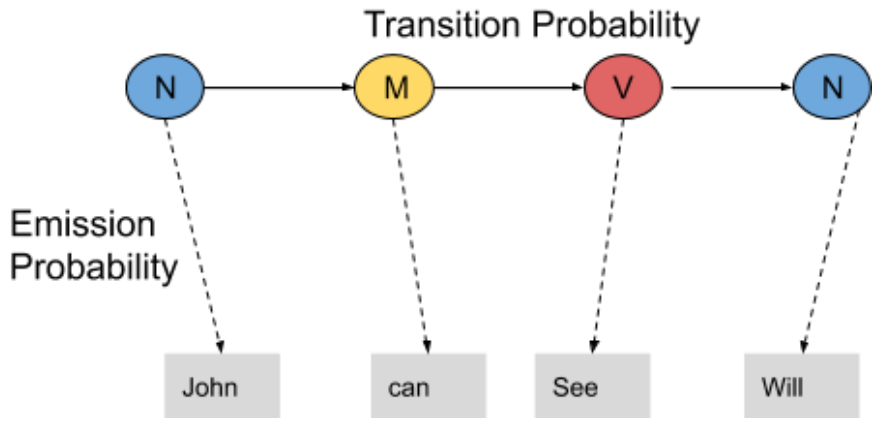
در بالاتر مسئله Speech tagging را معرفی کردیم، حال به شیوه حل کردن آن با HMM میپردازیم. همانطور که در شکل زیر میبینید اجزای جمله به چهار بخش تقسیم شده اند.

مشاهدات: خود واژه‌ها هستند مثل john, is, the, doing.

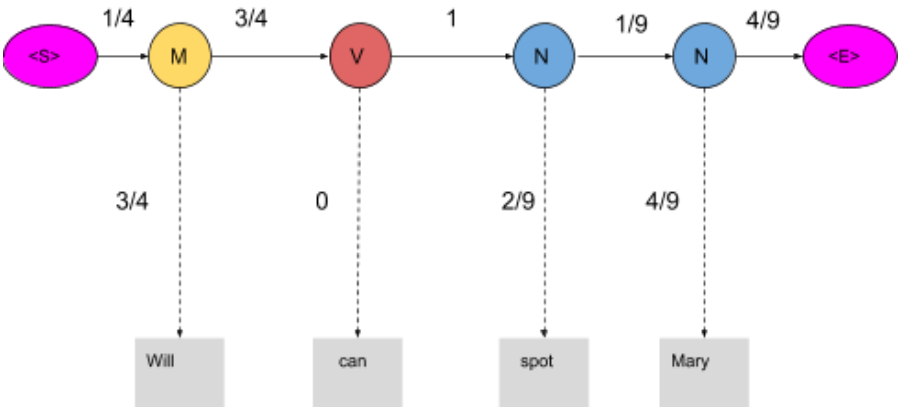
حالت‌ها: نقش‌های که اجزای به خود میپزیرند.

ماتریس حالات: احتمال اینکه یک نقش بعد از نقش دیگری بیاید. برای مثال احتمال اینکه فعل(verb) بعد از فاعل(subject) بیاید.

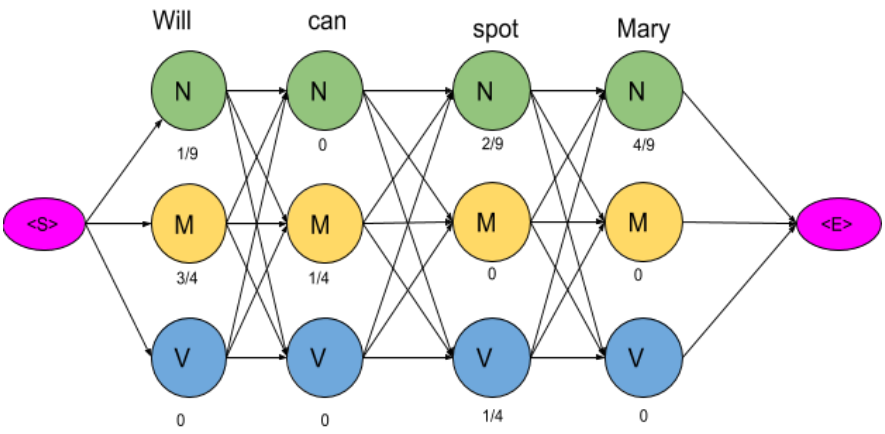
ماتریس مشاهدات: احتمال اینکه این واژه یک نقش به خصوصی را داشته باشد. مثلا احتمال اینکه 'john' را مشاهده کنیم به شرطی که حالت فعلی NOUN باشد.



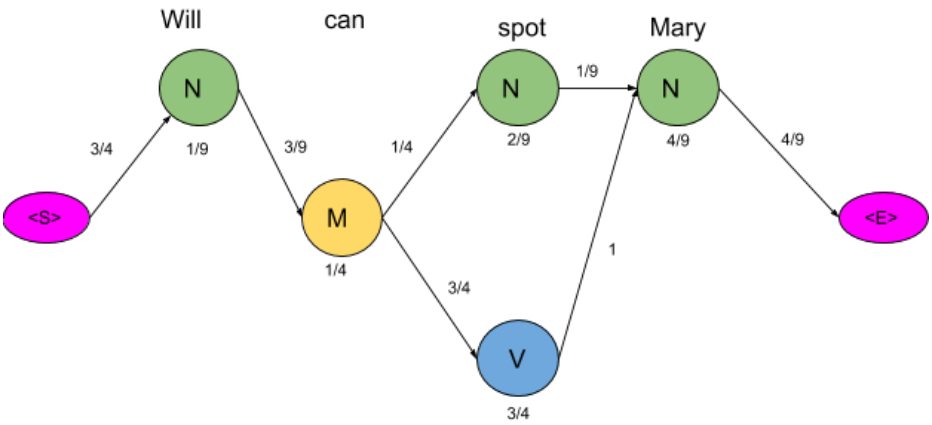
در زیر مثال واضح‌تری را مشاهده میکنیم. (<s> به معنی start یعنی شروع جمله است.)



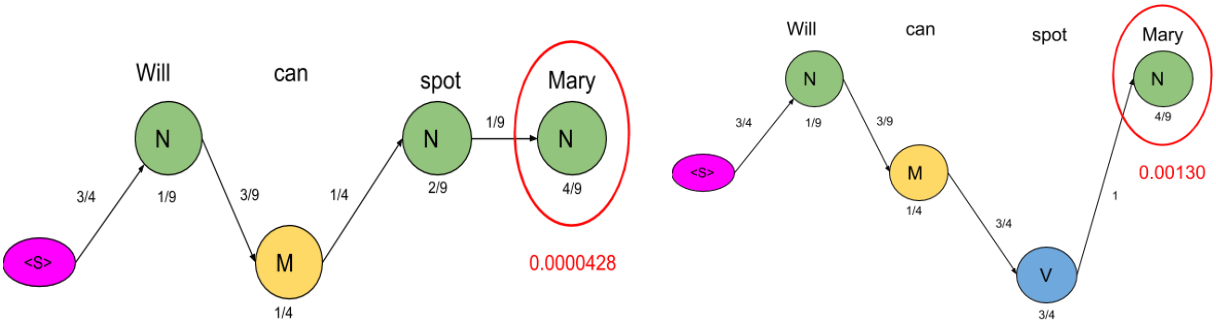
سپس الگوریتم گراف زیر را برای خود متصور میشود.



سپس با استفاده از Viterbi بهترین مسیری که واژه‌های این جمله را توصیف میکند پیدا میکنیم. ابتدا مانند شکل زیر تمام یال‌های که احتمال صفر دارند را حذف میکنیم.



سپس احتمال وقوع هر مسیر را حساب میکنیم.



بعد مسیری که احتمال وقوع بیشتری داشته است را برمیگردانیم که در این جا حالت دوم (سمت راستی) انتخاب میشود.

حالا به سراغ کدهای داخل Jupyter notebook میرویم.

داده‌های که به ما داده شده است در یک فایل txt به شکل رو به رو است پس ابتدا باید انها را مرتب کنیم.

```
i    PRP
'd   MD
like     VB
to   TO
go   VB
to   IN
a    DT
fancy    JJ
restaurant   NN
.    .

```

فرایند تبدیل داده چندان به مطالب این بخش مربوط نیست پس به آنها نمپردازیم ولی نتیجه تبدیل داده‌ها را در زیر می‌آوریم. همانطور که میبینیم داده‌ها به صورت یک لیست از دوتایی‌ها درآمده است که هر عضو لیست یک جمله که آنهم یک لیست از دوتایی‌های است که واژه و نقش آن را نگه داری میکنند.

```
[('please', 'UH'),
 ('give', 'VB'),
 ('me', 'PRP'),
 ('a', 'DT'),
 ('place', 'NN'),
 ('where', 'WRB'),
 ('i', 'PRP'),
 ('can', 'MD'),
 ('eat', 'VB'),
 ('excellent', 'JJ'),
 ('salad', 'NN'),
 ('plates', 'NNS'),
 ('.', '.')]
```

حال به تعریف چند تابع میپردازیم که این توابع داده‌ها را اماده خورده شدن به کتابخانه مورد نظر میکنند.

در زیر تابع rearrange_data را میبینیم که مجموعه train یا test ما را به عنوان ورودی میگیرد و چهار متغیر به ما برمیگرداند. اولی و دومی به ترتیب لیستی از واژه‌ها و نقش‌های‌شان هستند و سومی و چهارمی نیز مجموعه‌ای از واژه‌ها و نقش‌های‌شان هستند. (فرق مجموعه به لیست این است که در مجموعه هیچوقت یک عضو دو بار تکرار نمیشود که به ما در ساختن ماتریس‌ها کمک میکند.)

```
def rearrange_data(sequences):
    x = []
    y = []
    w = set()
    t = set()
    for sequence in sequences:
        sequence_x = []
        sequence_y = []
        for word, tag in sequence:
            sequence_x.append(word)
            sequence_y.append(tag)
            w.add(word)
            t.add(tag)
        x.append(sequence_x)
        y.append(sequence_y)
    return x, y, w, t
```

```
split = int( len(list_of_lists_of_tuples) * 0.9)
train = list_of_lists_of_tuples[:split]
test  = list_of_lists_of_tuples[split:]

train_x, train_y, train_words, train_tagset = rearrange_data(train)
test_x, test_y, test_words, test_tagset = rearrange_data(test)
```

تابع replace_unknown تمام واژه‌های ناشناخته‌ای که در مجموعه واژگان ما نیستند را با ’nan‘ جانشین میکند. این تابع کار توابع بعدی را ساده تر میکند.

تابع simplify_decoding با استفاده از الگوریتم Viterbi بهترین مسیر را مشخص و برمیگرداند.

```
def replace_unknown(sequence, vocabulary):
    """Return a copy of the input sequence where each unknown word is replaced
    by the literal string value 'nan'. Pomegranate will ignore these values
    during computation.
    """
    return [w if w in vocabulary else 'nan' for w in sequence]

def simplify_decoding(X, model, vocabulary):
    """X should be a 1-D sequence of observations for the model to predict"""
    _, state_path = model.viterbi(replace_unknown(X, vocabulary))
    return [state[1].name for state in state_path[1:-1]]  # do not show the start/end state predictions
```

در تابع accuracy تعداد تشخیص‌های درست مدل را حساب کرده و از طریق آن دقت الگوریتم را بر حسب درصد حساب میکند.

```
def accuracy(X, Y, model, vocabulary):
    """Calculate the prediction accuracy by using the model to decode each sequence
    in the input X and comparing the prediction with the true labels in Y.

    The X should be an array whose first dimension is the number of sentences to test,
    and each element of the array should be an iterable of the words in the sequence.
    The arrays X and Y should have the exact same shape.

    X = [("See", "Spot", "run"), ("Run", "Spot", "run", "fast"), ...]
    Y = [(), (), ...]
    """
    correct = total_predictions = 0
    for observations, actual_tags in zip(X, Y):

        # The model.viterbi call in simplify_decoding will return None if the HMM
        # raises an error (for example, if a test sentence contains a word that
        # is out of vocabulary for the training set). Any exception counts the
        # full sentence as an error (which makes this a conservative estimate).
        try:
            most_likely_tags = simplify_decoding(observations, model, vocabulary)
            correct += sum(p == t for p, t in zip(most_likely_tags, actual_tags))
        except:
            pass
        total_predictions += len(observations)
    return correct / total_predictions
```

در تابع pair_counts تعداد دفعاتی که یک واژه یک نقش به خصوص را گرفته است حساب میکند و برمیگرداند. از این عدد برای ساختن ماتریس مشاهدات استفاده میشود.

```
def pair_counts(sequences_A, sequences_B):
    """Return a dictionary keyed to each unique value in the first sequence list
    that counts the number of occurrences of the corresponding value from the
    second sequences list.

    For example, if sequences_A is tags and sequences_B is the corresponding
    words, then if 1244 sequences contain the word "time" tagged as a NOUN, then
    you should return a dictionary such that pair_counts[NOUN][time] == 1244
    """
    counts = defaultdict(Counter)
    for tags, words in zip(sequences_A, sequences_B):
        for tag, word in zip(tags, words):
            counts[tag][word] += 1
    return counts
```

تابع unigram_counts تعداد دفعات رخداد یک نقش در ورودی را حساب میکند. از این عدد برای ساختن ماتریس انتقال کمک میگیریم.

```
def unigram_counts(sequences):
    """Return a dictionary keyed to each unique value in the input sequence list that
    counts the number of occurrences of the value in the sequences list. The sequences
    collection should be a 2-dimensional array.

    For example, if the tag NOUN appears 275558 times over all the input sequences,
    then you should return a dictionary such that your_unigram_counts[NOUN] == 275558.
    """
    counts = Counter()
    for sequence in sequences:
        for tag in sequence:
            counts[tag] += 1
    return counts
```

تابع bigram_counts تعداد دفعاتی که دو نقش پشت‌سرهم در جمله آمده‌اند را حساب میکند. ازین عدد برای ساختن ماتریس انتقال کمک میگیریم. میتوان برای سه نقش هم این کار را انجام داد ولی مقالات مشخص کرده‌اند تاثیر چندانی نخواهد داشت و تعداد این سر سه تایی‌ها و و چهارتایی‌ها و ... به صورت نمایی زیاد میشودند که حافظه زیادی را از ما میگیرد.

```
def bigram_counts(sequences):
    """Return a dictionary keyed to each unique PAIR of values in the input sequences
    list that counts the number of occurrences of pair in the sequences list. The input
    should be a 2-dimensional array.

    For example, if the pair of tags (NOUN, VERB) appear 61582 times, then you should
    return a dictionary such that your_bigram_counts[(NOUN, VERB)] == 61582
    """
    counts = Counter()
    for sequence in sequences:
        for tag1, tag2 in zip(sequence[:-1], sequence[1:]):
            counts[(tag1, tag2)] += 1
    return counts
```

در تابع starting_counts تعداد دفعاتی که یک نقش خاص جمله را شروع کرده است میشمارد. این عدد در ساختن ماتریس Pi به ما کمک میکند.

```
def starting_counts(sequences):
    """Return a dictionary keyed to each unique value in the input sequences list
    that counts the number of occurrences where that value is at the beginning of
    a sequence.

    For example, if 8093 sequences start with NOUN, then you should return a
    dictionary such that your_starting_counts[NOUN] == 8093
    """
    counts = Counter()
    for sequence in sequences:
        counts[sequence[0]] += 1
    return counts
```

در تابع ending_counts تعداد دفعاتی که یک نقش خاص به جمله پایان میدهد را میشمارد. از این عدد در ساخت ماتریس جدیدی به اسم ماتریس اتمام استفاده میکنیم.

```
def ending_counts(sequences):
    """Return a dictionary keyed to each unique value in the input sequences list
    that counts the number of occurrences where that value is at the end of
    a sequence.

    For example, if 18 sequences end with DET, then you should return a
    dictionary such that your_starting_counts[DET] == 18
    """
    counts = Counter()
    for sequence in sequences:
        counts[sequence[-1]] += 1
    return counts
```

حالا به قسمت اصلی کار میرسیم که کتابخانه pomegranate این کار را برای ما انجام میدهد. کتابخانه را نصب و include میکنیم.

ابتدا یک نمونه HMM میسازیم و سپس ماتریس مشاهدات و ماتریس انتقال و ماتریس شروع و ماتریس پایان را به آن اضافه میکنیم.

(تابع assert شرط روبه‌رویش را بررسی میکند و اگر false در جلوی آن قرار گرفت یک error برمیگرداند.)

(تابع DiscreteDistribution یک تابع توضیع احتمال است که توضیعی از اعداد قابل شمارش مثبت است و سپس نرمال شده است.)

تابع Bake() مدل را نهایی کرده و ماتریس خلوت(sparse) آن را میسازد و سپس تمام ماتریس‌ها را نرمال میکند و اطلاعات مربوط به توضیع‌ها را ذخیره میکند edge ها را میسازد بهینه‌سازی های را هم انجام میدهد.

```
#pip install pomegranate
```

```
from pomegranate import State, HiddenMarkovModel, DiscreteDistribution
```

```
model = HiddenMarkovModel(name="hmm-tagger")

states = dict()
for tag, words in emission_counts.items():
    n = tag_unigrams[tag]
    assert n == sum(words.values())
    probs = {w:c / n for w, c in words.items()}
    emissions = DiscreteDistribution(probs)
    state = State(emissions, name=tag)
    model.add_states(state)
    states[tag] = state

n = sum(tag_starts.values())
for tag, counts in tag_starts.items():
    model.add_transition(model.start, states[tag], counts / n)

for (tag1, tag2), counts in tag_bigrams.items():
    model.add_transition(states[tag1], states[tag2], counts / tag_unigrams[tag1])

for tag, counts in tag_ends.items():
    model.add_transition(states[tag], model.end, counts / tag_unigrams[tag])

model.bake()

print('Edges', model.edge_count())
```

```
Edges 570
```

تابع bake یک گراف نیز می‌سازد که بیان‌کننده انتقال بین حالت‌ها است که در این مثال گراف ما 570 رأس دارد.

و در نهایت دقت مدل (ارزیابی عملکرد) را روی داده‌های آموزشی و تست میگیریم که نتایج در زیر آمده است.

```
#train_words = list(set(list_of_words))
training_acc = accuracy(train_x, train_y, model, vocabulary=train_words)
print("training accuracy: {:.2f}%".format(100 * training_acc))

testing_acc = accuracy(test_x, test_y, model, vocabulary=train_words)
print("testing accuracy: {:.2f}%".format(100 * testing_acc))
```

```
training accuracy: 96.37%
testing accuracy: 96.03%
```

قسمت سوم:

حال سعی میکنیم مقداری از کدهای این کتابخانه را خودمان بنویسم که البته به کیفیت کتابخانه نخواهد شد چون به دلیل کوچک بودن مقیاس این پروژه به درستی بهینه‌سازی نشده است.

همانند قسمت دوم داده‌ها را از فایل txt بیرون کشیده و به صورت لیستی از جمله ها درمی‌آوریم و سپس داده‌ها را به دو مجموعه آموزش و تست تقسیم میکنیم.

```
# compute Emission Probability
def word_given_tag(word, tag, train_bag = tuple_data):
    tag_list = [pair for pair in train_bag if pair[1]==tag]
    count_tag = len(tag_list)#total number of times the passed tag occurred in train_bag
    w_given_tag_list = [pair[0] for pair in tag_list if pair[0]==word]
    # calculate the total number of times the passed word occurred as the passed tag.
    count_w_given_tag = len(w_given_tag_list)

    return (count_w_given_tag, count_tag)
```

تابع word_given_tag یک کلمه و یک نقش را به عنوان ورودی میگیرد و تعداد رخ دادن نقش را مشمارد، سپس تعداد دفعاتی که این کلمه آن نقش را گرفته هم میشمارد و سپس این دو عدد را برمیگرداند. این تابع برای ساختن ماتریس مشاهدات است.

```
# compute  Transition Probability
def t2_given_t1(t2, t1, train_bag = tuple_data):
    tags = [pair[1] for pair in train_bag]
    count_t1 = len([t for t in tags if t==t1])
    count_t2_t1 = 0
    for index in range(len(tags)-1):
        if tags[index]==t1 and tags[index+1] == t2:
            count_t2_t1 += 1
    return (count_t2_t1, count_t1)
```

تابع t2_given_t1 تعداد دفعاتی که نقش t2 بعد از نقش t1 آمده است را میشمارد. این تابع برای ساختن ماتریس انتقال است.

```
# creating t x t transition matrix of tags, t= no of tags
# Matrix(i, j) represents P(jth tag after the ith tag)

tags_matrix = np.zeros((len(tags), len(tags)), dtype='float32')
for i, t1 in enumerate(list(tags)):
    for j, t2 in enumerate(list(tags)):
        temp = t2_given_t1(t2, t1)
        tags_matrix[i, j] = temp[0]/temp[1]

print(tags_matrix)
```

```
[[0.09255294 0.         0.         ... 0.02188913 0.00975494 0.1684511 ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 ...
 [0.         0.         0.         ... 0.00262697 0.         0.08756568]
 [0.00713012 0.         0.         ... 0.         0.00950683 0.        ]
 [0.00625579 0.00030893 0.00123571 ... 0.00084955 0.01019462 0.00648749]]
```

```
# convert the matrix to a df for better readability
#the table is same as the transition table shown in section 3 of article
tags_df = pd.DataFrame(tags_matrix, columns = list(tags), index=list(tags))
display(tags_df)
```

در این قسمت خود ماتریس انتقال را با استفاده از تابع t2_given_t1 ساخته‌ایم و سپس با استفاده از کتابخانه Pandas آن را به شکل زیباتر و خواناتری نمایش داده‌ایم که در فایل اصلی قابل مشاهده است.

```
def Viterbi(words, train_bag = tuple_data):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)

        pmax = max(p)
        # getting state for which probability is maximum
        state_max = T[p.index(pmax)]
        state.append(state_max)
    return list(zip(words, state))
```

در تابع Viterbi محتمل‌ترین مسیر را قرار است پیدا کنیم. ابتدا یک حلقه روی واژه‌ها میزنیم و چون اولین واژه، واژه قبلی ندارد قبل آن را نقطه میگزاریم. سپس از با توجه به حالت فعلی و حالت قبلی احتمال مورد نظر را از ماتریس احتمالات بیرون میکشیم و سپس با تابع word_given_tag احتمال مشاهده را هم حساب میکنیم و سپس با ضرب این دو احتمال و پیدا کردن حاصل‌ضربی که بیشترین مقدار را دارد و تکرار این کار مسیر بهینه را پیدا میکنم.

توجه: اجرای این تابع از نظر زمانی اصلا بهینه نیست زیرا که ما ماتریس مشاهدات را نساختیم و برای هر واژه جداگانه نسبت‌ها را پیدا میکنیم که این کار باعث هدر رفتن زمان بسیاری میشود. (این تابع از قصد اینگونه پیاده‌سازی شده است تا اهمیت پیاده‌سازی را در عملکرد بیان کند.)

نکته دیگر این که چون اجرای این تابع زمان زیادی میبرد فقط تعداد محدودی از جملات مجموعه تست را برای ارزیابی عملکرد انتخاب میکنیم.

```
#Here We will only test 10 sentences to check the accuracy
#as testing the whole training set takes huge amount of time
start = time.time()
tagged_seq = Viterbi(test_run_untagged_words)
end = time.time()
difference = end-start

print("Time taken in seconds: ", difference)

# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]

accuracy = len(check)/len(tagged_seq)
print('Viterbi Algorithm Accuracy: ',accuracy*100)
```

```
Time taken in seconds:  593.4353129863739
Viterbi Algorithm Accuracy:  94.52679589509692
```

همانطور که میبینیم فقط برای 100 جمله حدود 600 ثانیه یا 10 دقیقه زمان برده است و دقت آن هم به خوبی کتابخانه قسمت دوم نیست که دلیل این موضوع را در بخش بعدی گزارش بررسی میکنیم.

```
# To improve the performance,we specify a rule base tagger for unknown words
# specify patterns for tagging
# Although this is a very simple version and it might do noting.
patterns = [
    (r'.*ing$', 'VBG'),
    (r'.*ed$', 'VBD'),
    (r'.*es$', 'VBZ'),
    (r'.*\'s$', 'PRP'),
    (r'.*s$', 'NNS'),
    (r'.*', 'NN')
]

# rule based tagger
rule_based_tagger = nltk.RegexpTagger(patterns)
```

در قسمت معرفی زمینه‌کار گفتیم که روش‌های hybrid عملکرد بهتری نسبت روش‌های آماری و Rule-based دارند. در اینجا قصد داریم یکی از این روش‌ها معرفی و پیاده‌سازی کنیم.

در اینجا مثال بسیار ساده‌ای از این مسئله آورده شده است زیرا مثال‌های پیچیده‌تر نیاز به دانش تخصصی در مورد زبان هدف دارد که اینجا موضوع بحث ما نیست.

در قسمت قبل دیدیم که تمام واژه‌های ناشناخته را با 'nan' جایگذاری کردیم که عملا داریم قسمتی از اطلاعات را از بین میبریم درصورتی که میشود از آنها استفاده کرد به این صورت که با استفاده از regular expression ها میتوانی تشخیص دهیم ایا این واژه ناشناخته یک شماره تلفن، آدرس ایمیل، آدرس IP و یا یک فعل ماضی، مضارع یا حال استمراری یا یک اسم جمع و یا .... همانطور که گفتیم این اطلاعات بی‌ارزش نبوده و قابل استفاده هستند پس هرگاه در الگوریتم به یک واژها ناشناخته برخوردیم با از این روش کمک میگیریم. در نتیجه این کار باید الگوریتم Viterbi را دوباره نویسی کنیم.

```
#modified Viterbi to include rule based tagger in it
def Viterbi_rule_based(words, train_bag = train_set_tuples):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)

        pmax = max(p)
        state_max = rule_based_tagger.tag([word])[0][1]


        if(pmax==0):
            state_max = rule_based_tagger.tag([word])[0][1] # assign based on rule based tagger
        else:
            if state_max != 'X':
                # getting state for which probability is maximum
                state_max = T[p.index(pmax)]


        state.append(state_max)
    return list(zip(words, state))
```

تنها تفاوت با الگوریتم قبلی این است که اگر حاصل ضرب احتمال صفر باشد از نتیجه به دست آمده از rule_based_tagger استفاده میکنیم.

```
#test accuracy on subset of test data
start = time.time()
tagged_seq = Viterbi_rule_based(test_run_untagged_words)
end = time.time()
difference = end-start

print("Time taken in seconds: ", difference)

# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]

accuracy = len(check)/len(tagged_seq)
print('Viterbi Algorithm Accuracy: ',accuracy*100)
```

```
Time taken in seconds:  575.9059166908264
Viterbi Algorithm Accuracy:  94.4127708095781
```

در اینجا نتیجه ارزیابی عملکرد الگوریتم را مشاهده میکنیم ولی در بخش بعدی گزارش دلیل این اتفاق را بیان میکنیم.

نتایج و تحلیل نتایج:

قسمت اول: حل POS tagging با کتابخانه pomegranate

در قسمت قبل دیدم که دقت مدل در حل کردن این مسئله روی داده‌های تست برابر 96.03٪ است که یک درصد کمتر از انتظار ما یعنی 97٪ بود که با توجه به دلایلی که بیان میکنیم منطقی هم هست.

در بسیاری از مقالات که نتایج 97٪ به بالا گرفته اند از یکی از پایگاه داده brown corpus یا wallstreet Journal استفاده کرده‌اند که برای مثال brown corpus از 500 موضوع و منبع مختلف جمع‌آوری شده است و که شامل بیش از شصت هزار جمله با ۴۵ tag مختلف است در صورتی که corpus ما حدود پانزده هزار جمله و ۳۶ tag مختلف است. دلیل اهمیت این موضوع این است که هرچه پایگاه جملات متنوع‌تر باشد تعداد صفرهای که در ماتریس خلوت (sparse) داریم کمتر میشود که یعنی تعداد مسیرهای بیشتری را در نظر میگیریم و هرچه تعداد tag ها بیشتر باشد یعنی واژه‌ها با دقت بیشتری دسته‌بندی شده اند و این تفاوت قائل شدن بین بعضی دسته از واژه ها اطلاعات بیشتری را در اختیار ما میگذارد.

مورد بعدی که به بالاتر بودن نتایج در مقالات کمک کرده است روش‌های smoothing برای HMM است. در این روش نه تنها به آنچه قبل واژه فعلی آمده توجه میکنیم بلکه به آنچه در بعد آن هم آمده است توجه میکنیم. البته این smoothing مربوط به مدل‌های مارکوو است؛ ما در جا دیگری نیز از smoothing داریم که برای رفع کردن خلوت بودن ماتریس استفاده میکنیم به اینگونه که از احتمالات غیر صفر مقداری میکاهیم و مقداری به احتمال‌های صفر می‌افزاییم. یکی از smoother های معروف Laplace smoothing نام دارد که در شکل زیر روش کار آن را میبینیم. (عکس سمت راست بعد از smoothing است.)

$$q(s|u,v) = \frac{c(u,v,s)}{c(u,v)} \qquad q(s|u,v) = \frac{c(u,v,s)+\lambda}{c(u,v)+\lambda V}$$

$$e(x|s) = \frac{c(s \rightsquigarrow x)}{c(s)} \qquad e(x|s) = \frac{c(s \rightsquigarrow x)+\lambda}{c(s)+\lambda V}$$

روش‌های مختلفی برای smooth کردن یک ماتریس خلوت وجود دارد که مفید بودن خود را ثابت کرده‌اند که ما در پروژه از آنها بهره‌ای نگرفتیم.

قسمت دوم: حل POS tagging با توابعی که ساختیم

در این قسمت به دقت 94.4٪ رسیدیم و از نظر زمانی هم بهینه نبود که علاوه بر دلیل‌های قسمت اول این بخش دلیل‌های دیگری هم دارد که در زیر بیان میکنیم.

دلیل اینکه زمان زیادی برای حل طول کشید این بود که الگوریتم Viterbi ماتریس مشاهدات (Observation matrix) را نداشت و برای هر واژه و هر نقش یک بار کامل پایگاه جملات را میخواند تا احتمال‌های مورد نیاز را پیدا کند که یعنی ۳۶ بار برای هر واژه و ما صد جمله داشتیم و اگر میانگین هر جمله ۵ واژه داشته باشد و ما ۱۰۰ جمله داشتیم پس در مجموع تقریبا ۱۸۰۰۰ بار پایگاه جملات را خوانده‌ایم که اهمیت بهینه‌سازی الگوریتم را در نتایج بیان میکند. البته باید به این موضوع اشاره کرد که این روش چندان بدون کاربرد نیست و در محیط‌های که از حافظه کمی برخوردار هستیم با حذف کردن ماتریس مشاهدات مشکل ما را حل میکند، برای مثال در brown corpus بالغ بر یک میلیارد واژه وجود دارد و ۴۵ tag مختلف را میتوان برای آنها در نظر گرفت که در این صورت ماتریس بسیار بزرگ و بسیار خلوتی خواهیم داشت.

دلیل دیگر پایین‌تر بودن دقت عدم وجود الگوریتم Baum-welch است. ما این مسئله را فقط با Viterbi حل کردیم ولی این الگوریتم Baum-welch است که میگوید ما چقدر با بیشینه محلی فاصله داریم (در اینجا حدود ۱.۶ درصد) و ماتریس‌های ما را بهینه‌سازی میکند.

اگر توجه کنیم میبینیم که مدل hybrid زمان کمتری را صرف کرده است و این همان زمانی است که به واژه‌های ناشناخته را به جای حل کردن با ماتریس‌ها با regular expression حل کردیم پس نتیجه میگیریم که روش‌های hybrid هم دقت را بالاتر و هم زمان اجرا را پایین‌تر میاورند.

جمع‌بندی و نتیجه‌گیری:

ما در این پروژه ابتدا مدل‌های پنهان مارکوو را بیان کردیم و الگوریتم‌های Forward-Backward و Viterbi و Baum-Welch را به صورت Vectorize شده پیاده سازی کردیم و سپس مسئله part of speech tagging را بیان کردیم و آن را به دو صورت حل کردیم یک بار با استفاده از کتابخانه‌های آماده و یک بار با توابعی که خودمان نوشتیم و نتایج به دست آمده آن را با سایر مقالات مقایسه کردیم و شباهت‌ها و تفاوت‌ها و دلایل هر یک را توضیح دادیم. مسئله POS مسئله حل‌شده ای تلقی میشود ولی اینجا هدف ما از این مسئله اثبات قدرت مدل‌های پنهان مارکوو و روش‌های بهتر کردن الگوریتم‌های آن از نظر زمان و حافظه و دقت را بیان کردیم و چند روش برای بهینه کردن عملکرد آن در شرایط مختلف را نیز اشاره داشتیم و به این نتیجه رسیدیم که HMM ها روش بسیار مناسب و سریع و باکیفیتی برای حل مسائل مربوط به سیستم‌های زمانی که به صورت تصادفی تغییر حالت میدهند(از جمله POS tagging) است به شرطی که هم در پیاده‌سازی هوشمندانه عمل کنیم و هم از HMM smoothing، Sparse Matrix smoothing و غیره و همچنین روش‌های hybrid سازی درست استفاده کنیم.

فهرست منابع و مراجع:

https://math.stackexchange.com/questions/1072607/what-are-filtering-and-smoothing-with-regards-to-hidden-markov-models

https://link.springer.com/article/10.1007/s11831-020-09422-4

https://link.springer.com/article/10.1007%2Fs42044-020-00063-1

https://en.wikipedia.org/wiki/Part-of-speech_tagging

https://en.wikipedia.org/wiki/Markov_model

https://en.wikipedia.org/wiki/Hidden_Markov_model

https://www.tutorialspoint.com/natural_language_processing/natural_language_processing_part_of_speech_tagging.htm

https://www.freecodecamp.org/news/an-introduction-to-part-of-speech-tagging-and-the-hidden-/markov-model-953d45338f24

/https://www.mygreatlearning.com/blog/pos-tagging

chapter 15 of Artificial Intelligence of Russell & Norvig

مقالات زیر هم استفاده شده‌اند که میتوانید آن‌ها را در پوشه "منابع(مقالات و کتاب‌ها)" از داخل فایل پروژه مطالعه نمایید.

A HMM Text Classification Model with Learning Capacity - A. SEARA VIEIRA

A Machine Learning Approach to POS Tagging - LLUˊIS M`ARQUE

A Systematic Review of Hidden Markov Models and Their Applications - Bhavya Mor

Content Based Text Classification Using Morkov Models - Khalid Hussain Zargar

A SIMPLE RULE-BASED PART OF SPEECH TAGGER - Eric Brill

Some Advances in Transformation-Based Part of Speech Tagging - Eric Brill 1994

Improving Part-of-Speech Tagging for NLP Pipelines - Vishaal Jatav, Ravi Teja

Maximum Entropy Model for Part-Of-Speech Tagging - Adwait Ratnaparkhi

POS Tagging A Machine Learning Approach based on Decision Trees - Lluís Marquez i Villodre

Part-of-Speech Tagging from 97% to 100% - Is It Time for Some Linguistics - Christopher D. Manning

Part-Of-Speech Tagging using Neural network - Ankur Parikh

PART-OF-SPEECH TAGGING WITH NEURAL NETWORKS - Hehnut Schmid - Hehnut Schmid

POS with HMM-Assuming Joint Independence – Sang Zoo Lee

Sequence Labeling for Parts of Speech and Named Entities - Daniel Jurafsky

The Computational Complexity of Rule-Based Part-of-Speech Tagging - Karel Oliva

To Normalize, or Not to Normalize. The Impact of Normalization on POS Tagging - Rob van der Goo

tutorial on hmm and applications – Lawrence R. Rabiner

A Second-Order Hidden Markov Model for Part-of-Speech Tagging - Scott M. Thede

Hidden Markov Model- Based Korean POS tagging considering high agglutinativeity -sang-zoo Lee 2000

Unsupervised Part-Of-Speech Tagging with Anchor Hidden Markov Models – Karl Stratos

Different Approaches to Unknown Words in a Hidden Markov Model POS Tagger -Martin Haulrich