گزارش کار تمرین کامپیوتری2

کدینگ منبع:

بخش 1: طراحی تابعی برای اجرای الگوریتم هافمن و انکود کردن رشته دریافتی

در الگوریتم هافمن هریک ازکاراکتر ها احتمال وقوعی دارد.برای محاسبه codeword برای هر کاراکتر لازم است در ایتدا کاراکتر ها را بر اساس احتمال وقوع آن ها مرتب کنیم سپس دو احتمالی که از همه کوچک تر هستند را جمع کرده و برابر احتمال جدیدی قرار می دهیم و سپس احتمال های موجود را که در حال حاضر از تعداد آن ها یکی کم شده است را دوباره مرتب کرده و عملیات قبلی را روی آن اجرا می کنیم. این کار را تا جایی ادامه می دهیم که تعداد احتمال های موجود برابر یک شوند.سپس به هر دوشاخه ای ایجاد شده بیت های 0 و 1 را نسبت می دهیم و به این وسیله codeword های هر کاراکتر را به دست می آوریم.

برای اجرای این الگوریتم کلاسی به نام HuffTree در نظر گرفته شده است که هر object ای که از آن ساخته می شود نشان دهنده یک کاراکتر، احتمال وقوع آن و شاخه ی سمت چپ و شاخه ی سمت راست می باشد که این شاخه ها برای راحتی همان zero و one در نظر گرفته شده اند که در ابتدا None هستند.

کاراکتر ها و احتمال وقوع هر یک از آن ها در دیکشنری ای به نام weight_dict نگه داری می شوند.

تابعی به نام combine در نظر گرفته شده که با گرفتن دو object از کلاس HuffTree احتمال وقوع آن دو را باهم جمع کرده و یک objectجدید از کلاس HuffTree می سازد که دارای نماد کاراکتر نیست ولی احتمال وقوع آن جمع احتمال وقوع دو object قبلی است و شاخه های سمت راست و چپ آن همان object های قبلی خواهند بود.

توسط اجرای تابع weight_dict_to_tree_nodes ، به ازای هر کاراکتر موجود درweight_dict یک object از کلاس HuffTree ساخته شده وبه آرایه ای به نام nodes اضافه می شود. در نتیجه nodes آرایه ای از object های کلاس HuffTree می باشد.

در تابع build می خواهیم درخت هافمن این آرایه ایجادشده(nodes) بسازیم. به این منظور آرایه را مرتب می کنیم و دو عنصر آن را که دارای احتمال وقوع کم تری هستند را خارج کرده و روی آن دو تابع combine را صدا می کنیم و نود return شده از تابع combine را به عنوان نود جدید به آرایه اضافه می کنیم.و دوباره تابع build زا به صورت بازگشتی فراخوانی می کنیم تا زمانی که اندازه آرایه nodes به یک برسد و آن را برمی گردانیم (که نشان دهنده ریشه در درخت هافمن می باشد)

در تابع make_huffman_tree دو تابع توضیح داده شده در بالا به ترتیب فراخوانی می شوند وپس از اجرای این تابع درخت هافمن را به طور کامل در اختیار داریم.

برای تخصیص codeword به هر کاراکتر لازم است که از ریشه درخت ساخته شده در مرحله قبل شروع کنیم و بیت های 0 و 1 را به هر شاخه اختصاص دهیم به این منظور از تابع huffman_coding استفاده شده است که در آن ابتدا رشته ای تهی در نظر گرفته می شود. سپس ایت تابع برای شاخه ی سمت چپ و شاخه ی سمت راست هر نود به صورت بازگشتی فراخوانی می شود و در هر فراخوانی در شاخه سمت چپ (zero)به رشته مورد نظر "0" و در شاخه سمت راست (one) به رشته مورد نظر "1" اضافه می شود. در آخر دیکشنری ای به نام code_dict داریم که حاوی کاراکترها و codeword های آن هاست.

در این بخش می خواهیم رشته ای از کاراکتر ها را دریافت کرده و آن ها را encode کنیم پس با در اختیار داشتن codeword مربوط به هر کاراکتر و آگاهی به این که رشته دریافتی شامل چه کاراکتر هایی است و در کنار هم قرار دادن codeword های مربوط به آنان، میتوانیم رشته ای حاوی 0 و 1 برگردانیم. تابع return_huffman_code همین کار را انجام می دهد.

بخش 2: دیکود کردن رشته دریافتی

در این مرحله می خواهیم رشته ای از 0 و1 ها را دریافت کرده و رشته ای حاوی کاراکتر های زبان انگلیسی را برگردانیم

تابع مربوط به این قسمت تابعی به نام huffman_decoding است . در پیاده سازی این تابع در ابتدا رشته ای تهی در نظر گرفته شده که در هر مرحله که جلو می رویم یعنی به ازای هر 0 و 1 در رشته دریافتی کاراکتر نود ریشه را به رشته مورد نظر اضافه می کنیم. در هر مرحله اگه بیت در حال بررسی برابر 0 بود، ریشه را آپدیت کرده و ریشه ی جدید رو میزاریم نود شاخه zero و اگر 1 بود ریشه ی جدید رو میزاریم نود شاخه one و سپس سراغ بیت بعدی در رشته ورودی می رویم.

کدینگ کانال:

بخش اول: کدینگ Convolutional

این بخش به وسیله state machine پیاده سازی شده است. به این منظور یک دیکشنری به نام conv_state_machine در نظرمیگیریم که شامل فیلد های zero,one,two,three می باشد که نشان دهنده شماره state هاست. هر استیت خودش شامل دیکشنری ای است که دارای دو فیلد 0 و 1 است که به منظور نمایش حالت هایی است که یک transition می تواند داشته باشد

هر کدام از این فیلد ها خودش یک دیکشنری است که شامل فیلد هایی برای نگه داری استیت بعدی و parity تولید شده در آن مرحله است(این parity ها 2 بیتی می باشند . در نتیجه به ازای هر بیت ورودی دو بیت تولید می شود و در نتیجه طول رشته نهایی تولید شده دو برابر طول رشته ورودی است)

با کمک تابع Convolutional_encode_st از استیت zero شروع می کنیم و در هر مرحله با توجه به کاراکتر رشته ورودی در آن مرحله و استیتی که در آن هستیم مشخص می شود چه parity تولید می شود و به کدام استیت می رویم. در نتیجه با در کنار هم قرار دادن parity ها در هر مرحله رشته نهایی با طول دو برابر رشته ورودی تولید می شود.

بخش دوم: الگوریتم Viterbi

برای پیاده سازی این الگوریتم از روش Dynamic Programming استفاده شده است. به این منظور در هر زمان به هر استیت یعنی استیت های zero,one,two,three یک مقدار metric نسبت داده شده است که در ابتدا مقدار این metric برای استیت zero برابر 0 و برای بقیه استیت ها برابر بی نهایت در نظر گرفته شده است زیرا که حتما باید از استیت 0 شروع کنیم.

برای مشخص کردن مقدار metric ها در هر مرحله کافی است مقدار metric را برای همه استیت ها در مرحله قبل داشته باشیم و هم چنین تعداد بیت های متفاوت بین داده ی دریافتی و داده ای که به بیشترین احتمال در مسیر از استیت اولیه تاکنون ارسال شده است را داشته باشیم . از آن جایی که به هر استیت از طریق دو استیت دیگر می توان وارد شد پس مقدار metric در هر استیت از روی مقدار metric دو استیت قبلی خود که می توان از آن ها به این استیت وارد شد و هم چنین داشتن تعداد بیت های متفاوت(HD) به دست می آید به این صورت که مقدار metric استیت های قبلی را را با تعداد بیت های متفاوت(HD) در آن استیت جمع کرده و بین دو حالت موجود مینیمم را انتخاب می کنیم. و همه ی این مراحل را برای پیدا کردن metric استیت ها در زمان های بعدی تکرار می کنیم.

در دیکشنری ای به نام state_machine فیلد های zero,one,two,three تعریف شده اند که برای هر رسیدن به هر کدام از این استیت ها می توان از دو branch متفاوت استفاده کرد که هر کدام از این branch ها حاوی اطلاعات زیر اند:

prev_st: نشان دهنده ی استیت قبلی است که از طریق آن دراین branch موردنظر به این استیت مورد نظر می رسیم به طور مثال در شکل داده شده نشان می دهد که به استیت zero می توان از استیت های zero یا one وارد شد پس برای استیت zero مقدار این فیلد در یکی از branch ها zero و در دیگری one است.

input_b: در استیت قبلی به ازای آمدن چه ورودی به این استیت می آییم

out_b: با ورود به استیت جدید از استیت قبلی چه خروجی تولید می شود.

آرایه ای دو بعدی از دیکشنری ها به نام PM در نظر گرفته شده است که سطر های آن نشان دهنده گذر زمان است که شامل دو سطر 0 و 1 می باشد چرا که برای اجرای این الگوریتم و به دست آوردن metric های جدید در هر مرحله فقط به مرحله قبل نیازمندیم. و ستون ها نشان دهنده استیت های zero,one,two,three می باشند.

لازم به ذکر است که بعد از دریافت ورودی آن را دو بیت دو بیت تقسیم کرده و آن را به تابع Viterbi می دهیم.

برای پیاده سازی این الگوریتم از تابع Viterbi استفاده شده است.

در هر مرحله سطر اول آرایه PM، (PM[0])که نشان دهنده مقدار metric های تمام استیت ها در مرحله قبلی است را داریم و کافی است که با توجه به الگوریتم توضیح داده شده مقدار metric را برای مرحله جدید محاسبه کنیم و در سطر دوم قرار می دهیم. در سطر دوم آرایه PM علاوه بر این که مقدار metric های جدید را نگه می داریم مقدار branch را هم نگه می داریم که بدانیم در هر مرحله از کدام branch استفاده کردیم که به وسیله آن رشته محتمل تر و ورودی را به وسیله داشتن branch و اطلاعات مربوط به آن در state machine به دست آوریم.

بعد از محاسبه سطر دوم آرایه PM کوچک ترین مقدار metric در آن مرحله را پیدا کرده و باتوجه به مقدار branch آن و اطلاعات state machine مقدار ورودی ای که به ازای آن این branch را انجام دادیم و هم چنین رشته ی تولید شده در این branch را به دست می آوریم.

بعد از آن برای محاسبه مقدار metric ها در مرحله بعد فقط به سطر دوم آرایه PM فعلی نیازمندیم پس آن را در سطر اول قرار داده و سطر دوم آرایه PM برای دریافت مقدار metric در زمان بعدی آماده است.

در آخر با کنار هم قرار دادن مقدار ورودی و رشته ی تولیدی که در هر مرحله به دست آمد می توان ورودی و رشته ای که به مقدار کمینه خطا اختصاص دارد را به دست آوریم.

بخش3 : بخش پایانی

در این بخش در ابتدا تابع make_huffman_tree صدا زده می شود که درخت هافمن را با روش گفته شده در بالا بسازد سپس تابع huffman_coding صدا زده می شود که با داشتن درخت ساخته شده codeword مربوط به کاراکترها را به دست آورده و دریک دیکشنری ذخیره می کند. سپس تابع return_huffman_code فراخوانی می شود که به عنوان آرگومان ورودی آن رشته نام و نام خانوادگی بدون space را می دهیم، خروجی این تابع رشته ای از 0 و 1 هاست که به وسیله ارجاع به دیکشنری مربوط به codeword هر کاراکتر به دست می آید.سپس رشته ی خروجی این تابع را به تابع Convolutional_encode_st می دهیم می دهیم که در این تابع به وسیله state machin در آن به ازای هر بیت ورودی ، دو بیت خروجی تولید می شود پس خروجی این تابع، رشته ای از 0 و 1 ها به اندازه

2 برابر طول رشته ورودی است. سپس خروجی این تابع را به تابع noise داده که به صورت رندم بعضی از بیت های ورودی را عوض می کند. سپس خروجی این تابع را به تابع Viterbi می دهیم که این تابع به وسیله سازوکار های داخل آن رشته ورودی به انکودر کانال و رشته ی ورودی ای که احتمالا بدون ارور بوده را محاسبه کرده و به عنوان خروجی باز می گرداند. سپس رشته خروجی این تابع را به دیکودر هافمن که همان دیکودر منبع است می دهیم و به وسیله دیکشنری ای که در ابتدا از codeword ساخته شد و همچنان آن را در اختیار داریم آن را دیکود می کنیم و رشته ای از کاراکتر های انگلیسی را بر می گردانیم. به دلیل تغییر بعضی از بیت ها بعد از عبور از تابع noise رشته خروجی دیکودر هافمن با رشته ورودی به انکودر هافمن(نام و نام خانوادگی بدون فاصله) دقیقا یکسان نخواهد بود.

جند نمونه از خروجی برنامه: