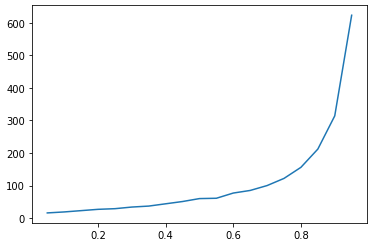
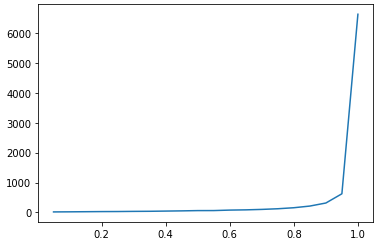
**PageRank**

Per l’implementazione del PageRank svolge un ruolo di primaria importanza la scelta dei parametri *k* e *s*, dove *k* rappresenta il numero di cicli di aggiornamento del rank, mentre *s* rappresenta lo scaling factor, in cui il valore s\*rank(x) viene suddiviso tra i vicini del nodo e il valore (1-s)\*rank(x) tra tutti i nodi del grafo.  
Si noti che, in quanto la sommatoria di tutti i rank è sempre pari ad 1, la suddivisione di (1-s)\*rank(x) tra tutti i nodi equivale ad un rank di partenza pari a (1-s)/n, con n numero di nodi, a cui successivamente vengono sommate le frazioni del rank dei rispetti vicini.

L’algoritmo è stato implementato in due versioni, nella prima sono stati utilizzati i dizionari per il calcolo del rank (*dict version*), mentre la seconda strutta il prodotto tra matrice e vettore (*matrix version*).  
L’Update Rule nella *dict version* è stata applicata secondo il ragionamento soprariportato, mentre nella *matrix version* ci si è trovati di fronte ad una problematica. La versione matriciale prevede *k* moltiplicazioni tra la trasposta di una particolare matrice e il vettore dei rank. Tale matrice si costruisce ricavandosi la matrice (sparsa) di adiacenza *M* del grafo stocastico di G, a cui viene poi applicata la formula s\*M+(1-s)/n, dove la somma di (1-s)/n viene applicata ad ogni elemento. Ciò comporta la costruzione di una matrice di dimensioni 22470x22470 con tutti gli elementi diversi da 0 che causa un troppo elevato utilizzo di memoria. Per risolvere tale problematica si è deciso di non costruire tale matrice ma di effettuare il prodotto della trasposta della matrice sparsa M con il vettore di rank per ottenere un vettore che successivamente viene moltiplicato per s, per in fine sommare (1-s)/n ad ogni elemento del nuovo vettore ottenuto.

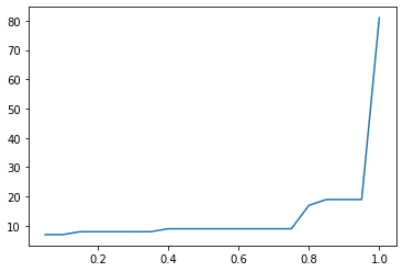
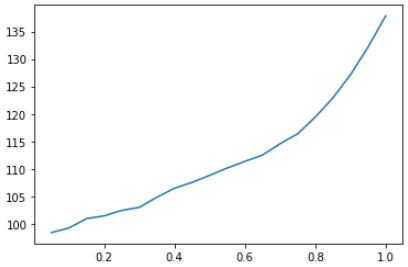
Le versioni sono risultate equivalenti in termini di cicli per ottenere la convergenza, con solo una leggera differenza tra i risultati. In particolare, gli elementi che compongono la top500 risultano rimanere invariati, ma sono presenti dei piccoli cambi di posizioni. Tali cambi di posizione sono probabilmente causati da una propagazione dell’errore durante i cicli di aggiornamento della *dict version*. Infatti, si è potuto verificare che nella *matrix version* la somma dei rank durante ogni ciclo risulta sempre pari ad 1, mentre nella *dict version* tale somma risulta diminuire con il procedere dell’algoritmo fino ad ottenere un valore di circa a 0.99 al suo termine. Un errore totale dell’ordine di 10-2 risulta essere molto elevato considerando che il valore medio dei rank è dell’ordine di 10-5.

Durante lo studio dei parametri s e k si è riscontrato che il numero di cicli *k* necessari per ottenere la convergenza dei valori dei rank risulta crescere all’aumentare di *s*. Per lo studio della convergenza è stata presa in considerazione la sommatoria dei valori assoluti delle differenze tra i rank al k-esimo e al (k-1)-esimo ciclo. In particolare, solo in pochi casi si è riusciti ad ottenere una differenza tra cicli consecutivi pari a 0, si suppone a causa di errori di macchina, mentre nella maggior parte dei casi tale valore risulta diminuire fino a valori limite (o pattern periodici) dell’ordine di 10-18 o inferiori. Si è considerata come convergenza la raggiunta di questo limite. Qui di seguito vengono mostrati due grafici che mostrano il numero di cicli di aggiornamento *k* necessari per arrivare a convergenza per ogni scaling factor multiplo di 0.05 compreso tra 0.05 e 1 nella prima immagine e compresi tra 0.05 e 0.95 nella seconda (per mostrare al meglio l’evoluzione per i primi valori di s).



I grafici mostrano come la relazione tra k e s sia di tipo esponenziale, con un notevole aumento della pendenza per valori superiori a 0.9 ed in particolare per s=1.

Successivamente si è passati all’analisi dei risultati. Per i vari valori di *s* esaminati, sono stati riscontrati 207 nodi presenti in ogni top500, anche se in differenti posizioni. Inoltre, si è potuto notare come valori maggiori di *s* tendano a dare uno score più elevato ai nodi con grado maggiore. Di seguito vengono riportati due grafici, in cui quello a sinistra mostra la media dei gradi dei nodi presenti nelle top500 e il grafico a destra mostra il grado minimo relativo ai nodi presenti nelle top500.



In particolare, si può notare come per s = 1, la top500 è formata esattamente dai 500 nodi con grado maggiore in quanto il grafo è connesso e non diretto e di conseguenza non sono presenti trappole.  
Inoltre, per valori di *s < 0.8* è stata riscontrata la presenza di nodi di grado inferiore a 10 che ci permette di concludere che bassi valori di *s* tendono a dare un alto score anche a nodi con pochi collegamenti, pur continuando a dare molta importanza anche a nodi di grado elevato, come messo in luce dal grafico della media, quasi sempre superiore a 100.

Il miglior trade-off dei parametri è stato quindi deciso in base a tali informazioni. Al fine di ottenere tempi di esecuzione brevi con risultati che non avvantaggino troppo nodi con pochi collegamenti, si è scelto come migliore trade-off il valore di *s* pari a 0.85 che necessita di un numero minimo di cicli *k* pari a 211 per convergere. Con tale valore, inoltre, la media dei gradi dei nodi nella top500 risulta essere 122.922, mentre il grado minimo 19. Il valore di *k* scelto per effettuare i successivi esperimenti è 250, per essere sicuri di arrivare a convergenza, tenendo conto che un aumento così piccolo dei cicli di aggiornamento non comporta grandi rallentamenti.

Una volta scelti i valori ottimali di s e k (0.85 e 250) si è passati allo studio dei tempi di esecuzione dei vari algoritmi proposti. Si precisa che per le implementazioni parallele delle due versioni è stata utilizzata una struttura dati condivisa tra i jobs, un dizionario nel caso della *dict version* e un vettore nella *matrix version*, data la necessità di dover conoscere in ogni thread e ad ogni ciclo i valori dei rank aggiornati dagli altri threads. Tale scelta ha permesso di ottenere risultati molto vicini a quelli delle versioni Naive, ma non uguali a causa di un non perfetto sincronismo tra i job che può causare un imperfetto aggiornamento della struttura dati e quindi una riduzione dell’accuratezza dell’algoritmo.

La *dict version* Naive impiega circa 5 minuti per la completa esecuzione dell’algoritmo, tempo accettabile ma non ottimo, in quanto la *matrix version* Naive risulta molto più veloce con un tempo di esecuzione di soli 10 secondi. Le relative versioni parallele, a causa della gestione dei lock molto onerosa, non permettono di migliorare tali tempi. Infatti, la *dict version* parallela risulta avere un tempo di esecuzione di circa 6 minuti con l’utilizzo di 4 job e di 5 minuti e 30 secondi con 2. La *matrix version* parallela, invece, pur risultando ugualmente molto veloce comporta un rallentamento di pochi secondi sia con l’utilizzo di 2 job che con 4.

La *matrix version* Naive risulta quindi essere sia la versione più veloce, con brevissimi tempi di esecuzioni, sia la più accurata in quanto le versioni parallele non riescono ad ottenere gli stessi risultati di tale versione, mentre la *dict version* Naive risulta essere affetta dal problema della propagazione dell’errore di calcolo.