SF5 Final Report

myl43

June 2024

1 Introduction

Collective brains keep innovation alive. In his book "The Secret of Our Success", Henrich [2] argues that humans progressed so rapidly not because of genetic evolution, but rather cultural evolution: our ability to socialize gives us access to stores of information far beyond what an individual can handle. A key feature of this collective brain is the spreading of skills, where the population as a whole learns to perform better through individuals learning from neighbours.

In order to illustrate this, he presents a simple network experiment, comparing a population of "Geniuses" and "Butterflies":

- Geniuses create an invention once in 10 lifetimes, but only have 1 friend.
- Butterflies are 100 times less inventive, creating an invention once in 1000 lifetimes, but have 10 friends.

If a friend has an invention, the individual has a 50% chance of learning it. After everyone has tried to invent something themselves and tried to learn from friends, the results are:

- Geniuses: 18% acquire the invention, with half inventing it themselves.
- Butterflies: 99.9% acquire the invention, but only 0.1% invent it themselves.

The size of a group and degree of social inter-connectedness is crucial in deciding how this effect takes place, and this simple experiment more or less demonstrates the usefulness of social connection. However, there are many ways to extend our investigation into the theory of cultural learning using math. The most obvious improvements to the simulation are making it closer to a real-world scenario, including setting a non-uniform starting connection distribution and modelling changing social networks. By looking at how the initial distribution of skill affects its propagation, then introducing more complex factors like noise and referral mechanisms, we can see how different network characteristics change the rate at which learning spreads, which allows us to get a deeper understanding of how we can promote the spreading of skills in a society.

2 Methods

2.1 Initialization

We can first create a more realistic model of social connections following a power law distribution, $P(k) \sim k^{-\gamma}$. Barbasi and Albert (1999) [1] pointed out that collaborations between actors followed such a distribution with gamma = 2.3, so we can use that as a helpful starting parameter. We generate the network with a configuration model, which basically allows us to set the degree of each node according to our desired distribution. We first generate a sequence of degrees, setting a min and max parameter: realistically, let's assume people at least start off with two connections, and cannot have more connections than the size of the population.

Each node is then assigned a parameter according to a set distribution: we will compare various initialization parameters, but will mainly assume these parameters are normally distributed. This parameter represents an individual's "ability". For now, we also assume the score is a linear function of the parameter. For our purposes, this means that we simply need to set the score equal to the parameter to capture representative behaviour.

2.2 Iterative updates

To model the learning process, we will allow each node to get information from its neighbours at each timestep, which we will call a generation. In this report, we will only look at one-sided learning, where nodes only try to copy neighbours that score higher as it is more realistic (justified in the appendix).

Every timestep, each node:

1. Calculates the difference between its scores and its neighbours scores
2. Calculates the difference between its parameter and its neighbours parameter for any neighbours that have a higher score
3. Updates its own score by adding the product of step 1 and 2 to its original score
4. We also assume there is a skill cap, and so the node truncates the updated score at 1 if past this boundary.

Mathematically, this means that each time-step we expect the parameter (θ) of node i to be:

$$\mathbb{E}[\theta_i(t+1)] = \theta_i(t) + \alpha\mathbb{E}\left[\sum_{j\in N(i), s_j > s_i} (s_j - s_i)(\theta_j(t) - \theta_i(t))\right]$$

where N(i) is the set of neighbours of node i, s_i is the score of node i, and α is the learning rate.

We can attempt to get an idea of how to calculate the second term on the right hand side if we take a simple network with a uniform (not power law) degree distribution, and a gaussian parameter initialization. Let X_1, X_2, X_3, X_4 be the **values** of four independent Gaussian distributions in order of size, randomly assigned to each node in this set of first degree connections including the node itself.

Because our initial distribution is independent, we can simplify the expected update for t = 1 to be

$$\frac{1}{4}(X_1 - X_2)^2 + \frac{1}{4}\left((X_1 - X_3)^2 + (X_2 - X_3)^2\right) + \frac{1}{4}\left((X_1 - X_4)^2 + (X_2 - X_4)^2 + (X_3 - X_4)^2\right)$$

Under independence, we can use symmetry to argue that every possibility (node is the highest among its neighbours, node is second highest, etc.) is the same, and we only care about the values of a set of four gaussians to get the expected update. However, once an update happens the values of each node are conditional on all its neighbours and so we end up with a sum of conditional probability for this node's

position in its neighbours scores multiplied by the conditional score of its neighbours given this position. As we progress, the dependency between nodes makes this very hard to calculate analytically, but we can see it is some function of the neighbour score variances.

2.3 Referral network

In later experiments, we extend these simulations to a dynamic network. The most intuitive way to model how a network changes with social learning is based off word-of-mouth: if someone you know is very good at something, you would be more likely to recommend a friend learn from them as well. To model this process, we run the following:

1. Every generation, we first calculate the scores of each node, then carry out the referral process for each node with a probability of referral_rate for each individual
2. if the referral process is started, the node finds its highest neighbour and a random neighbour
3. We randomly select a neighbour of the random neighbour to remove as a connection, and replace it with the highest neighbour of the original node

To try to understand the behaviour of the network, let's consider a very simple example of a uniformly distributed random network initialized with a Gaussian distribution. The expected value of a node's highest neighbor at the first generation is the maximum of three Gaussian-distributed variables. With probability referral rate β, the expected value of a random neighbor at $t = 2$ is:

$$\mathbb{E}[\theta_i^r(2)] = (1-\beta)\mathbb{E}[\theta_i(1)] + \beta f(\max(G_1, G_2, G_3), \theta_n(1))$$

for all $\theta_n(1)$ connected to $\theta_i(1)$ with a maximum degree of two.

This is necessarily very approximate and does not accurately reflect the simulations below (as a power law distribution would have different expected neighbours), but it gives us two useful insights. Firstly, the expected value of an average neighbour is dependent on the referral rate as time progresses. Secondly, the distribution of neighbors (variance and mean) is the other major factor in deciding the behaviour of the system, scaled by a factor of the referral rate. These two findings will be relevant in our discussion of dynamic referral networks after Figure 4.

3 Results & Discussion

3.1 Behaviour under changing network sizes and parameter initializations

A good place to start is trying to isolate one of the two main factors influencing how fast learning spreads: how connected and how big a network is. To do this, we can generate a "base" power-law degree distribution for the smallest network (here we use min degree of 2 and max degree 99). We then duplicate this "base" distribution as many times as we need to get the degrees for a larger network, e.g. a network of size 500 is just the original distribution repeated 5 times. This is used in a configuration model (basically a way to make a network with our specified degrees) to create the networks. Using these different sized networks, we assign each node a parameter (corresponding to ability at an arbitrary thing we want them to learn) according to one of four distributions with an approximate mean of 0.5: a uniform distribution, power law distribution, gaussian with low variance (s.d. = 0.1), and gaussian with high variance (s.d. = 0.5).

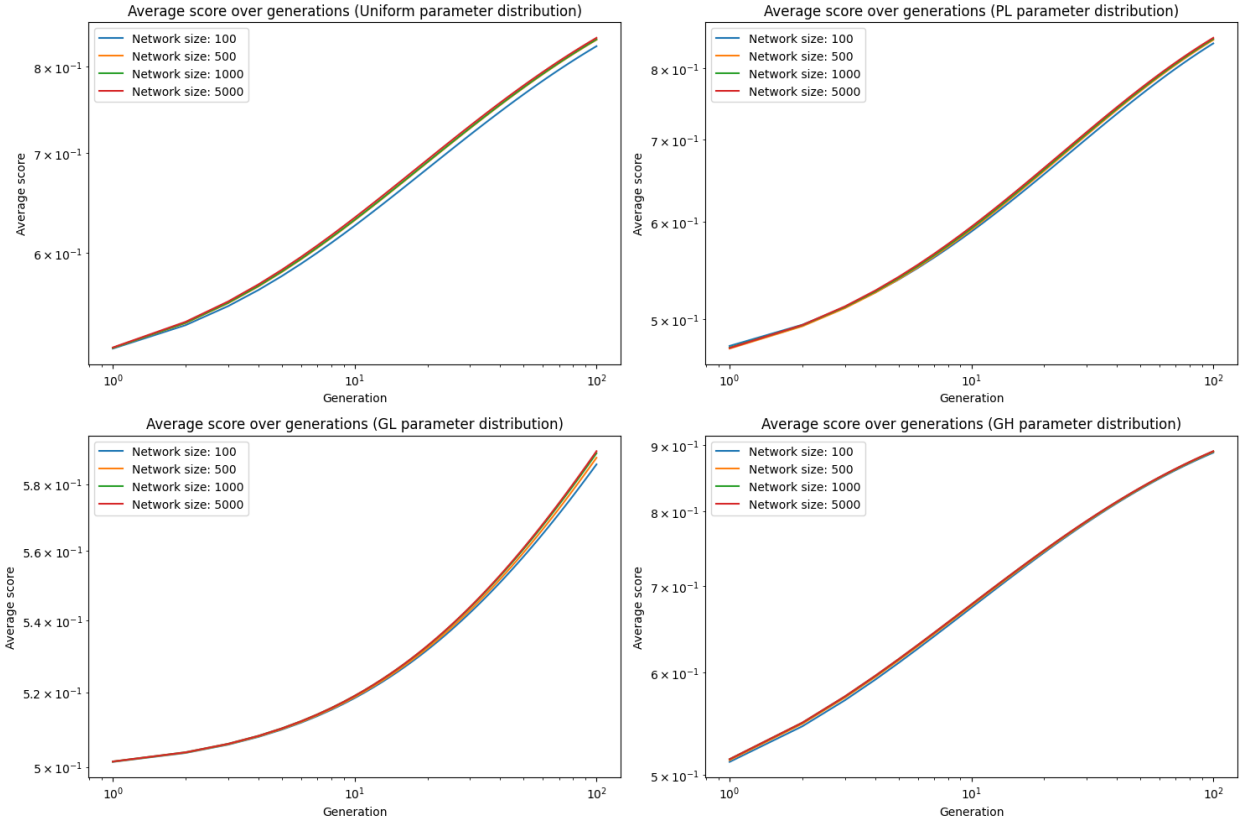


Figure 1: The effect of network size on score increase in power law distributed learning networks over different starting parameter distributions

In each simulation, the trend is relatively clear, with a larger network outperforming a smaller one: however, this difference is quite small relative to the overall score increases. Interestingly, if there is a low diversity in initial starting scores the distributions seem to diverge more, with a clear trend towards separating the lines. However, in the high gaussian noise case the lines appear to converge as we get close to the max value. A possible explanantion for this is that in general, nodes are connected to a more diverse sample in larger networks, which allows the learning from top performers to start affecting a larger proportion of nodes earlier in the spreading process.

We also notice that the curves are slightly different in each subplot of Figure 1. To try to find out why that happens, we can plot histograms of the population for a network of size 1000 in a single run to see

what's happening in the population. Note that in both figures we constrain the parameters to be between 0 and 1 after generating the initial parameters (which is why there are peaks at 0 and 1 for the high variance gaussian).

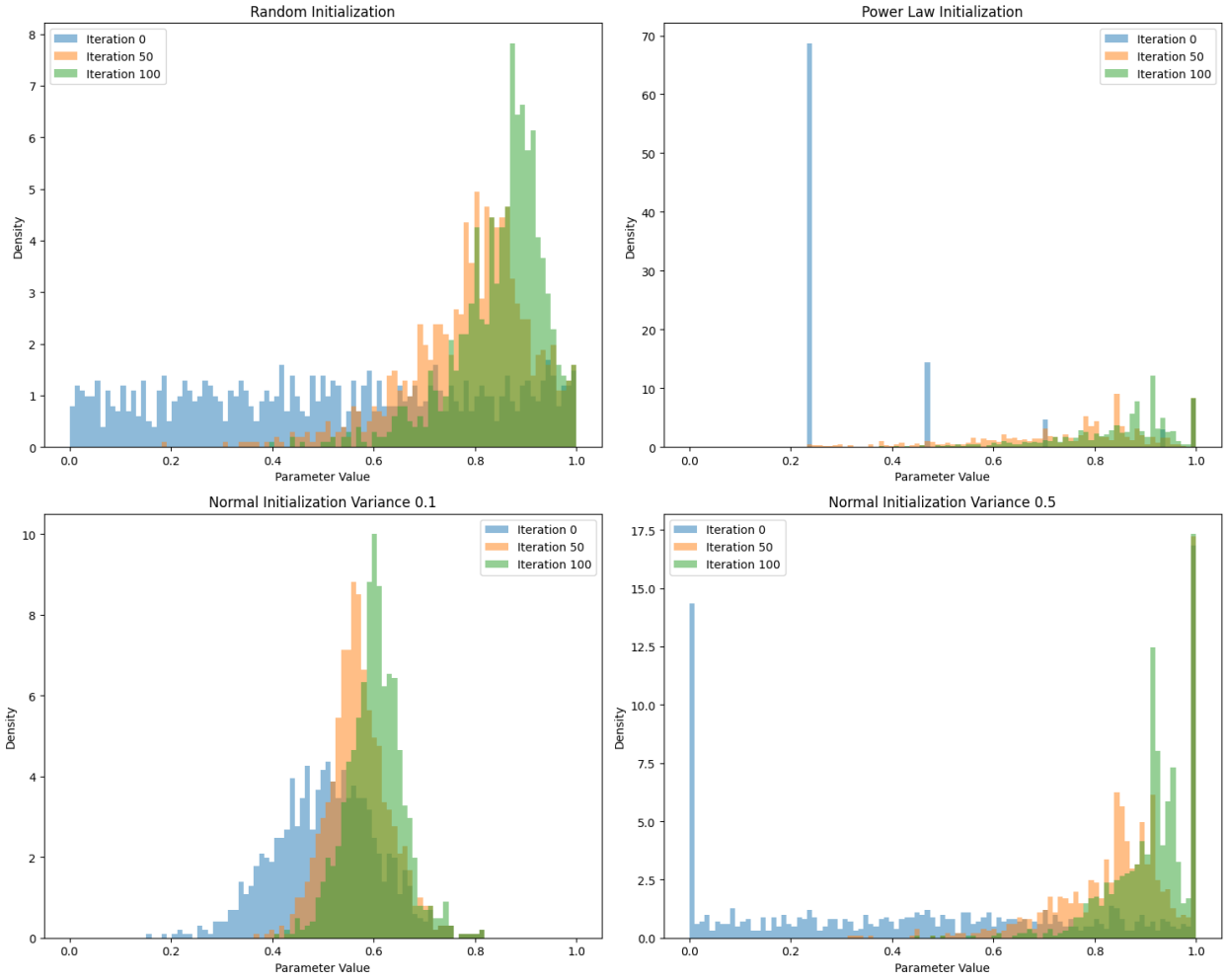


Figure 2: Evolution of population score distributions for different starting parameter distributions in a power law network

These histograms show us a few things:

1. The large low-scoring population in the gaussian high variance and power law diagrams are brought up to speed relatively quickly.

2. Below a certain score the rate of score increase seems to be going up even on a log-log scale, but tends towards a linear/even sloping down as values approach the maximum.

3. Even as the low variance gaussian seems to get more "compressed", the rate of average score increase is going up, probably because enough nodes have gone up for the network as a whole to be learning at a faster rate.

Compared to a uniform distribution, we see such a big difference in the final average scores of the high and low variance gaussians because of the difference in number of high-scoring individuals even when the mean is exactly the same. This effect is further highlighted by the fact that the power law distributed network performed similarly to the uniformly distributed network, showing that as long as there is a suitable concentration of highly skilled individuals, the network as a whole can make up for a large number of

under-performers as they can learn quickly.

This all assumes that performance in the real world is exactly reflective of ability, but how would network learning change when noise is introduced? To find that out, we assign the score of each node to be sampled from a gaussian distribution with a mean value equal to the ability parameter. We run this on a network of size 1000 for 1000 runs over 100 generations to try to even out the effect of variance.

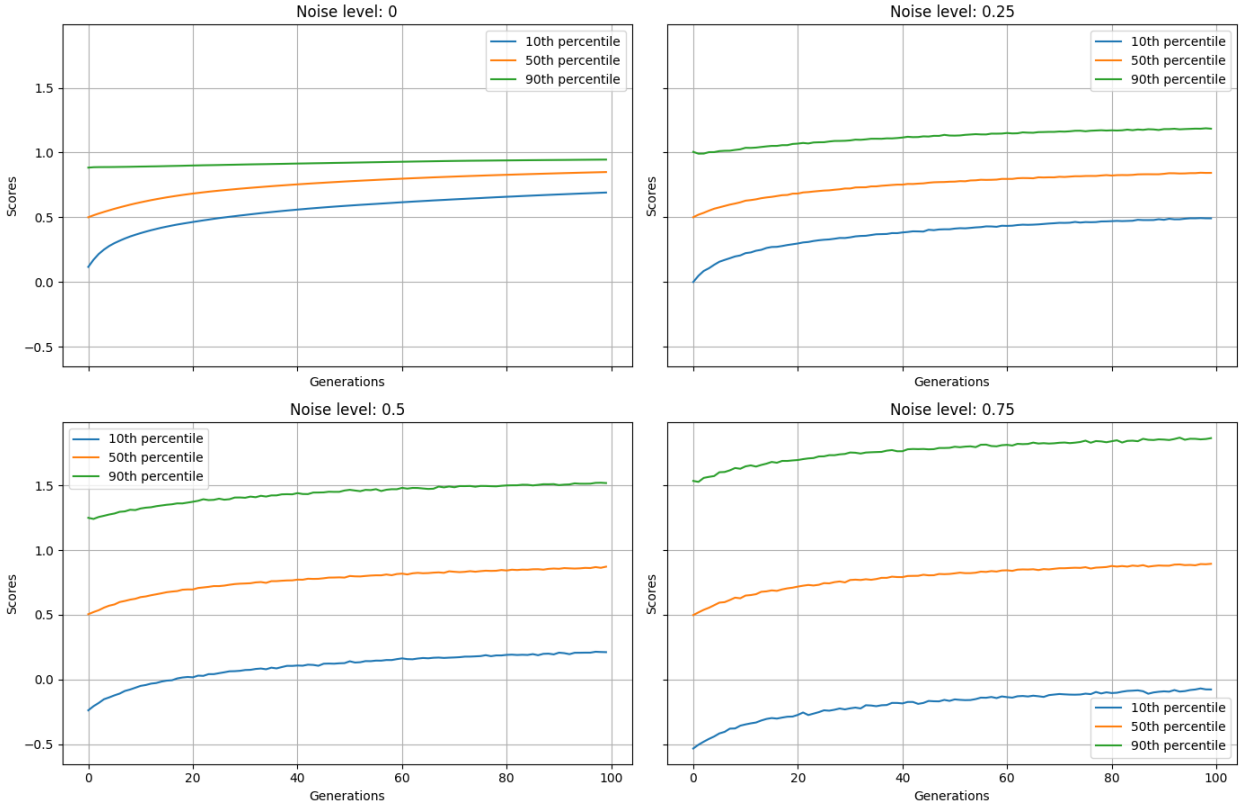


Figure 3: Average score percentiles after introducing varying levels of noise into a power law network

Intuitively, it would seem like noise makes learning harder: it is not as clear who to learn from, and people might actually regress if the noise is big enough. This explanantion seems to fit, with the average population score going down with more noise: however, the average score goes down less with incremental additions to the noise level. The plots show that as we increase the noise the percentiles diverge more, but because by random chance the top performers in a population far outperform the rest of the population, they actually balance out the effects of lagging behaviour. In other words, although noise definitely affects the rate at which the general population can learn things and widens the gap between the best and worst performers, it has a diminishing effect on the population's performance as a whole.

10th percentile	**50th percentile**	**90th percentile**	**Noise Level**
0.7403	0.8801	0.9551	0.25
0.4134	0.7475	1.0795	0.25
0.0830	0.7306	1.3765	0.5
-0.2258	0.7408	1.7052	0.75

Table 1: Score percentile values for network shown in Figure 3

3.2 Referral networks

Now that we've tested out some basic properties about learning in a network based on its initial skill distributions, it makes sense to also consider some extensions to the network structure itself. We could model situations like research conferences (temporarily increase connections in a cluster) or competing communities (densely connected islands), but the most general effect causing networks to change are probably related to our social interactions. Back to the book mentioned in our introduction, Henrich argues that learners use several cues to decide who to learn from: age, success, and prestige are the most obvious as they are indicators of a survival advantage. We're modelled success with scores, age is a bit harder to model unless we have a much more complicated model with birth and death, so prestige seems to be the factor that is both interesting and possible for us to explore. The simplest way to explore prestige is some measure of social success: we could model this as another score function, but the most obvious causal reason for prestige on average is success. In that case, a simple referral mechanism could get us quite far in understanding the effects of social recognition for success.

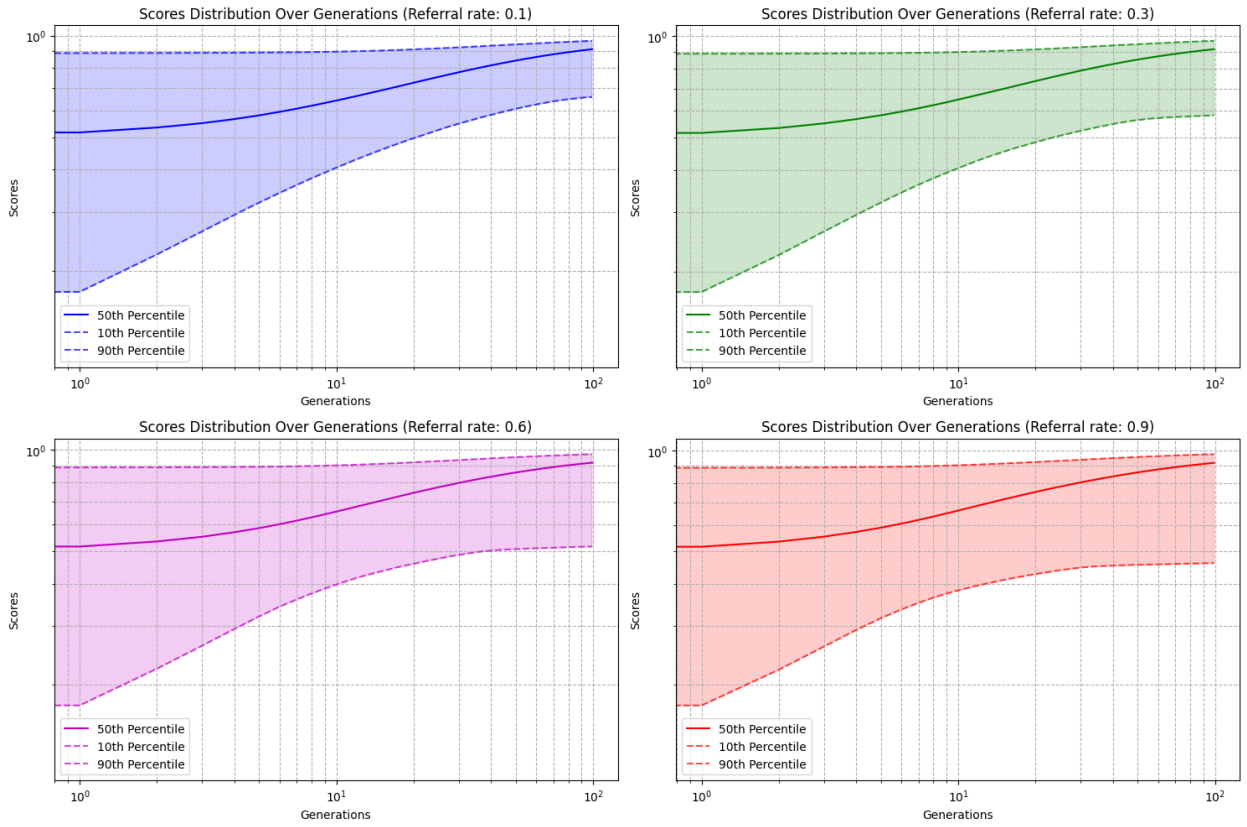


Figure 4: Log-log plots of population score percentiles for a network of size 5000 for different referral rates

The overall trend does appear the same as our initial network. However, what we're interested in here is the behaviour towards the skill boundary. This particular example is of a 5000-node network run for 100 generations and averaged over 50 runs, using an initial gaussian distribution with s.d = 0.3 here to speed up the evolution process. The lower 10th percentile follows a linear trend on the log-scale until it reaches approximately the 0.5 mark, at which growth starts to taper. A higher referral rate means that this tapering on the low end is more obvious as the worst-performing nodes end up more isolated. Throughout the generations the 10th percentile also grows at a slower rate for a higher referral rate. However, the average score of the population seems to still be similar, showing that the "stronger" candidates binding together more closely makes up for the lagging subset to a certain degree. The astute reader might at this point wonder what the tradeoff between these effects are, and if you look closely at the graphs you'll notice there is a relatively visible

point at which the growth curve slope shifts from increasing to decreasing around the 0.75 mark. The actual values taking the 50th percentile score at the 10-generation mark show that a referral rate of 0.9 outperforms the lower referral rate. However, if we look at the same measurement at 100 generations, this effect is flipped.

We can explore both of these effects by running several simulations on a smaller network: Figure 5 uses a network of size 500 with initialization using a 0.2 sd gaussian (This standard deviation was chosen as a tradeoff between speed of network evolution and how clear the distribution would look in Figure 6).

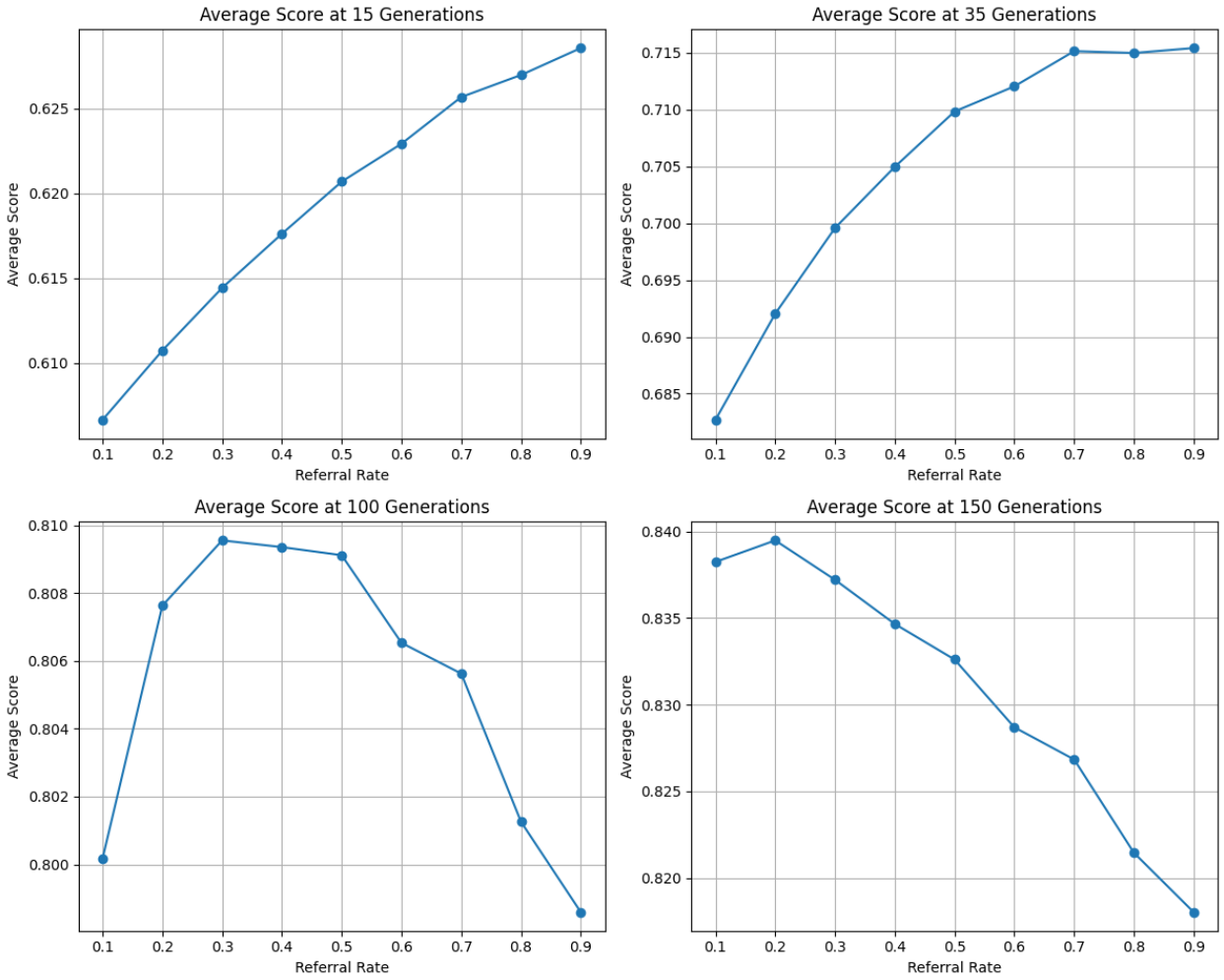


Figure 5: Average population score for different referral rates as the network progresses through generations

Taking "slices" of the average scores over 100 runs at empirically significant timepoints (15, 35, 100, 150 for a network of this size and initialization), we can observe a changing trend in the effect of the referral rate. At a lower average score, a lower referral rate (corresponding to a more evenly connected network) causes the population as a whole to improve slower; the more selectively concentrated networks with higher referral rates allow the high performers to learn rapidly and bring up the average. However, at a score around halfway between starting and max, the trend starts to reverse, resulting in an inverted "U" shape as a balance between boosting the top performers and ensuring connection diversity prevails. After this, when average scores start to approach the skill cap the isolated underperformers begin to be a proportionally bigger drag on the population performance, leading to a low referral rate (which enforced connections across skill levels) having a better performance overall.

We can see this in action by taking the same slices and plotting the full score distribution histograms for the two extreme values of referral rates. Throughout the simulation, it is clear that a 0.9 referral rate leads to a significantly larger population at the top end of skill, but the low tail of the 0.1 referral rate population gets pulled up much more.

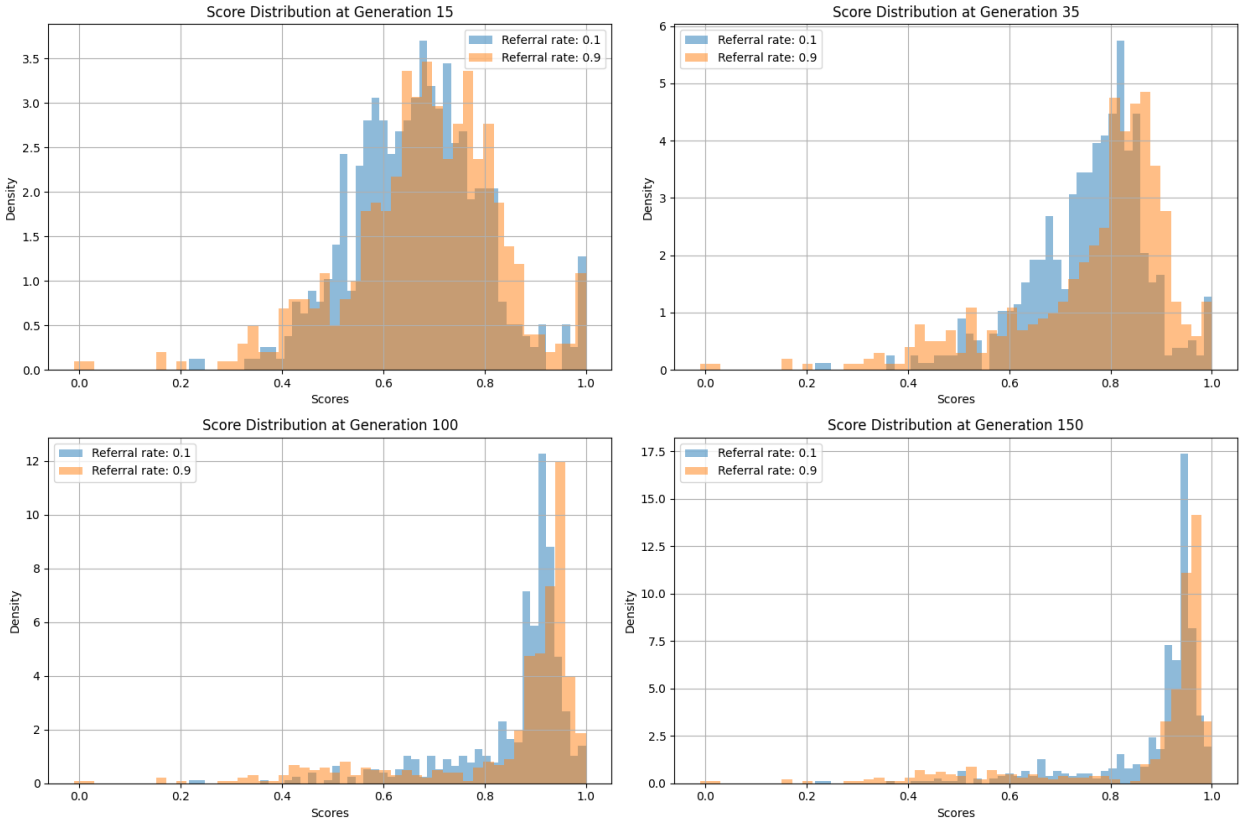


Figure 6: Score distributions for one run of a similar network to Figure 5, comparing the population at the same timepoints

This suggests at a very high level that depending on the current population skill level and distribution, a different level of attention to prestige may be optimal.

4 Conclusion

4.1 Overview

In the real world, we generally require the combination of several elements to come up with something new. As such, we can argue that the spread of a single piece of information is also representative of the development of innovation, as the rate of emergence of new inventions depends on slowest element the inventor learns. This preliminary study of learning propagation in power-law networks identifies the effect of connecting a diverse range of skill levels in improving population scores, shows that noise has a decreasing effect on the general ability of a population as a whole, and demonstrates the changing efficacy of biasing networks to connect to top-performing individuals.

I would argue that the two most interesting results were the fact that more noise doesn't strictly mean worse performance and that a higher referral rate is only beneficial in certain situations: the first result gives us some confidence that even in an environment where performance is heavily variant even under the same ability level, the population as a whole picks up on the elements that work; and that for the people who learn the wrong things and under-perform there are equally many who accidentally learn the right thing and do far better. The second result indicates that if we want a community as a whole to do well, we have to encourage interaction with all members - but when it is a group of unskilled individuals, it is better for the group as a whole to allow the best members to learn from one another, which helps pull up the average performance.

4.2 Limitations and extensions

Given that this report was relatively short, this should be treated much more as an exploratory piece of work, although the conclusions presented here should be valid by themselves in the restricted setting we've presented. The most obvious improvements start with running these simulations on larger networks for a wider range of situations - currently we've identified interesting directions with smaller networks and tried to go more in-depth along one path: to establish exact causality we need to simulate a much wider range and demonstrate where some phenomena do and do not happen. Enforcing a more mathematical approach to network initialization would also make these results more useful: a power law network should be an improvement over a uniform network in modelling social situations, but more thought into how we justify the parameters of the network to more closely reflect a real world learning society. Controlling the skill distribution depending on degree of an individual through seeding the network with set parameter values will additionally allow us to measure the out-sized effects that highly connected individuals have on the network - and although we compensate for that by average over runs, deliberately introducing this situation minimizes the problem.

There are many extensions to consider, primarily in translating empirical evidence from cultural observations into mathematical simulations. The simplest would probably be exploring the influence of prestige on learning in more detail - we can adjust the learning rate dynamically based on some arbitrary hidden value that measures how prestigious someone is, or try a more complex referral mechanism that has individuals choosing which connections to add/discard in a more systematic and nuanced way. The scoring system itself is also open to many improvements: we can create non-linear combinations of factors that result in success, which probably more accurately mirrors real life. A suitable function could be something along the lines of $f(x, y) = -x^2 - y^2$, with a clear maximum point but changing efficiencies of attributes in terms of achieving higher scores. Hidden variables would make this scoring system even more interesting, as people don't always share or know the exact reasons for their success, so we could for example carry out a simulation in which only a random subset of each node's characteristics are broadcast to its neighbours. Would the network as a whole be able to learn the appropriate set of skills by piecing together the individual components from different sources?

5 References

References

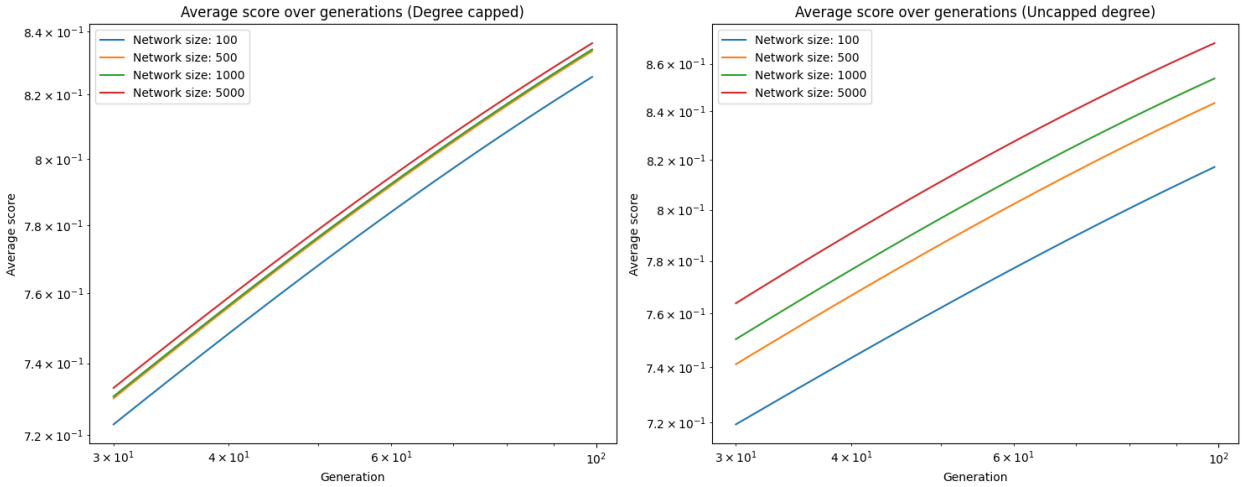
[1] R. Albert. "Emergence of Scaling in Random Networks". In: *Science* 286.5439 (1999). Source: PubMed, pp. 509–512. DOI: 10.1126/science.286.5439.509.

[2] Joseph Henrich. *The Secret of Our Success: How Culture is Driving Human Evolution, Domesticating Our Species, and Making Us Smarter*. Princeton, NJ: Princeton University Press, 2016.

6 Appendix

6.1 Unrestricted Degrees

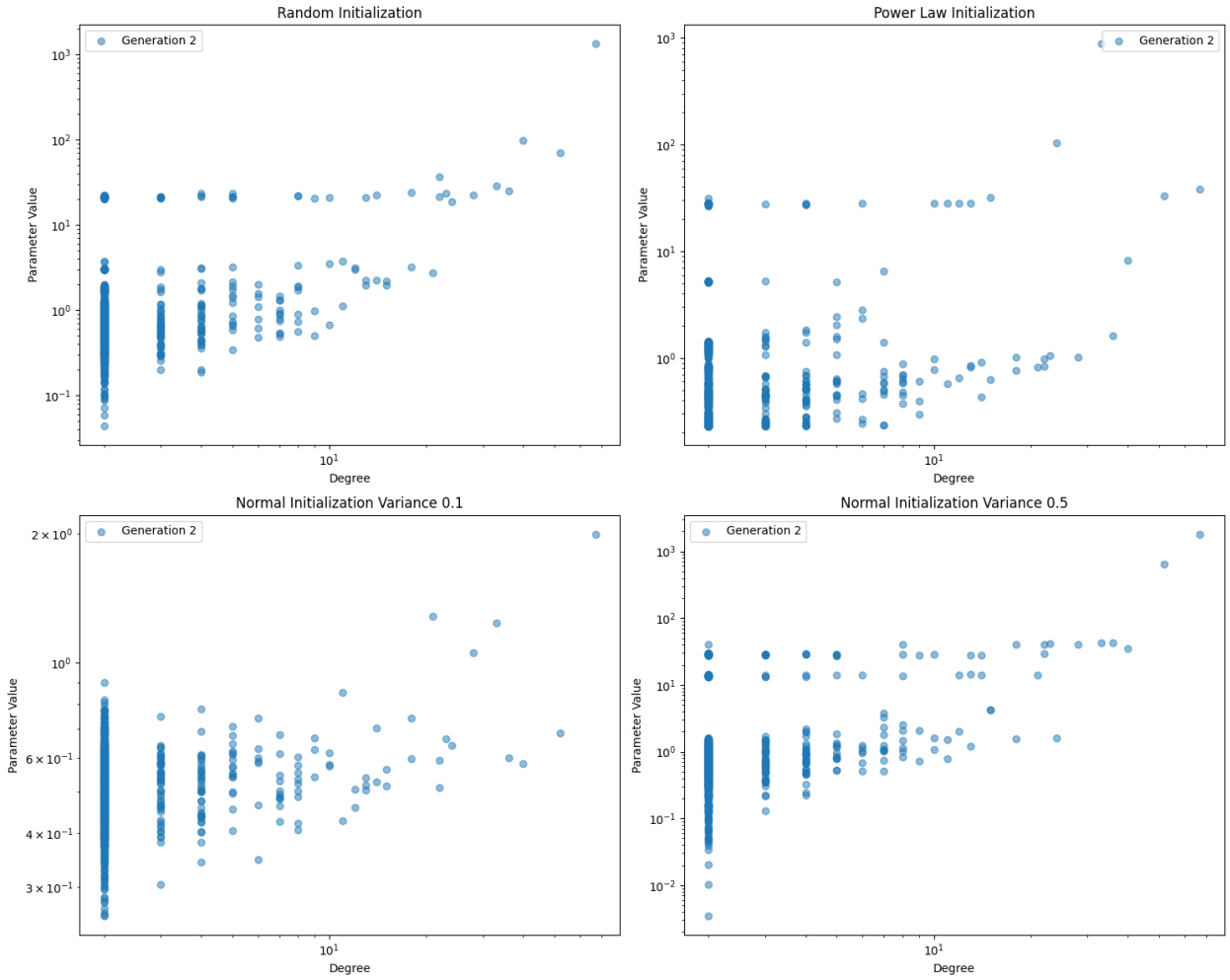
In figure 1, we mention the use of tiling: copying the same degree distribution for the different network sizes. The graphic below shows what happens if we don't do that: the mean degree only increases by around 0.3 with each step up, but the differences are obvious.



6.2 Two-way learning

The main reason for not using the two-sided learning model is that it results in unrealistic behaviour. This is most obvious when we remove the "skill cap" of 1 and the most connected node becoming very large within the simulation space we're interested in: we've used a power-law distributed learning model, and so just looking at the most connected node, every generation it will increase its score more than average nodes by an exponential factor (because every connection contributes to the score increase), with the rate only widening as the most connected node starts to outperform its neighbours by a large amount, further boosting its rate of increase.

Apart from the most connected node leading the infinite growth, any nodes it is connected to also spiral out of control. Given that we take the product of the score difference and parameter difference, this increase is incredibly rapid once the top performer exceeds 1 and when there is a difference bigger than 1. Below is a visualization of the degrees and parameter values on log scales for the same network shown in figure 2, but only at the second generation, when the values are already blowing up.



6.3 Code

To make it easier for reproduce and extend, a copy of the main notebook used here will be uploaded to *https* : *//github.com/MoseslGit/SF5_networks/*, but a cleaner version of the relevant code is written out below.

6.3.1 Network initialization

```
# Configuration model
def configuration_model(k):
    n = len(k)
    S = np.array([i for i in range(n) for _ in range(k[i])])
    S = np.random.permutation(S)
    if len(S) % 2:
        S = S[:-1]
    S = S.reshape(-1, 2)
    return S


# Sampling with power law distribution
def generate_power_law_degrees(n, gamma=2.3, min_degree=2, max_degree=None):
    if max_degree is None:
```

```
        max_degree = n - 1  # Set the maximum degree to n-1 by default

    # Use zipf distribution to generate power-law distributed degrees
    k = np.random.zipf(gamma, n)

    k = np.clip(k, min_degree, max_degree)

    # Total degree count must be even
    if np.sum(k) % 2 != 0:
        k[-1] += 1

    return k


def convert_to_adjacency_matrix(edges, n):
    A = np.zeros((n, n))
    for edge in edges:
        A[edge[0], edge[1]] = 1
        A[edge[1], edge[0]] = 1
    np.fill_diagonal(A, 0)
    return A
```

6.3.2 Score and parameter update functions

```
# Score calculation
def calculate_scores_equal(parameters):
    scores = parameters.copy()
    return scores

def calculate_scores_equal_noisy(parameters, noise=0.1):
    scores = parameters.copy()
    scores += np.random.normal(0, noise, scores.shape)
    return scores

# Efficient implementation of parameter update
def update_parameters_linear(scores, parameters, adjacency_matrix, learning_rate=0.1):
    # Expand scores to compare each node's score with its neighbors
    score_diff_matrix = adjacency_matrix * (scores - scores[:, np.newaxis])

    # set all negative values to 0
    score_diff_matrix = np.maximum(score_diff_matrix, 0)
    # Calculate relative parameters of successful and unsuccessful neighbors for each node
    relative_parameters = adjacency_matrix * (parameters - parameters[:, np.newaxis])

    # Elementwise multiplication of score_diff_matrix and relative_parameters
    update_param_weights = score_diff_matrix * relative_parameters

    # Sum the entire row to get the weight update for each node
    update_param_weights_sum = np.sum(update_param_weights, axis=1)

    # Update parameters
    updated_parameters = parameters + learning_rate * update_param_weights_sum
    # If updated parameters are over 1, set them to 1
    updated_parameters = np.minimum(updated_parameters, 1)
```

```
    return updated_parameters
```

6.3.3 Figure 1

```
network_sizes = [100, 500, 1000, 5000]
num_generations = 100
gamma = 2.3
num_runs = 50

def run_simulation_base(network_sizes, gamma, num_generations, num_runs,
restricted_degree=True, param_distribution='uniform'):
    average_scores_all_runs = {n: [] for n in network_sizes}
    mean_degrees_all_runs = {n: [] for n in network_sizes}

    # Repeat the operation for num_runs times
    for run in tqdm(range(num_runs)):
        if param_distribution == 'uniform':
            base_parameters = np.random.rand(network_sizes[0])
        elif param_distribution == 'power_law':
            # Adjusted by a factor of 4 to make the parameters
            # more similar to the uniform distribution
            base_parameters = np.random.zipf(2.3, size=network_sizes[0]) / 3.8
        elif param_distribution == 'gaussian_low_noise':
            base_parameters = np.random.normal(0.5, 0.1, network_sizes[0])
            base_parameters = np.clip(base_parameters, 0, 1)
        else:
            base_parameters = np.random.normal(0.5, 0.5, network_sizes[0])
            base_parameters = np.clip(base_parameters, 0, 1)

        base_degrees = generate_power_law_degrees(network_sizes[0], gamma)

        for n in network_sizes:
            if restricted_degree:
                # Restrict max_degree to 100 to aid comparison across network sizes
                degrees = np.tile(base_degrees, n//network_sizes[0])
            else:
                degrees = generate_power_law_degrees(n, gamma)

            mean_degrees_all_runs[n].append(np.mean(degrees))
            edges = configuration_model(degrees)
            A = convert_to_adjacency_matrix(edges, n)
            np.fill_diagonal(A, 0)

            # Make n/network_sizes[0] copies of the base parameters
            parameters = np.tile(base_parameters, n//network_sizes[0])
            scores = calculate_scores_equal(parameters)

            average_scores_n = []

            for i in range(num_generations):
                parameters = update_parameters_linear(scores, parameters, A)
                scores = calculate_scores_equal(parameters)
                average_scores_n.append(np.mean(scores))
```

```
            average_scores_all_runs[n].append(average_scores_n)

    # Compute the mean across all runs for each network size and generation
    final_average_scores = {n: np.mean(average_scores_all_runs[n], axis=0) for n in network_sizes}

    return final_average_scores, mean_degrees_all_runs

dc_final_average_scores, dc_mean_degrees_all_runs = run_simulation_base(network_sizes, gamma,
num_generations, num_runs)
dc_pl_final_average_scores, dc_pl_mean_degrees_all_runs = run_simulation_base(network_sizes, gamma,
num_generations, num_runs, param_distribution='power_law')
dc_gl_final_average_scores, dc_gl_mean_degrees_all_runs = run_simulation_base(network_sizes, gamma,
num_generations, num_runs, param_distribution='gaussian_low_noise')
dc_gh_final_average_scores, dc_gh_noise_mean_degrees_all_runs = run_simulation_base(network_sizes,
gamma, num_generations, num_runs, param_distribution='gaussian_high_noise')
fig, axs = plt.subplots(2, 2, figsize=(15, 10))
datasets = [dc_final_average_scores, dc_pl_final_average_scores, dc_gl_final_average_scores, dc_gh_final
titles = ['Uniform parameter distribution', 'PL parameter distribution', 'GL parameter distribution', 'G

for ax, data, title in zip(axs.flat, datasets, titles):
    for n in network_sizes:
        ax.plot(range(1, num_generations + 1), data[n], label=f'Network size: {n}')
    ax.set_xlabel('Generation')
    ax.set_ylabel('Average score')
    ax.set_xscale('log')
    ax.set_yscale('log')
    ax.set_title(f'Average score over generations ({title})')
    ax.legend()

plt.tight_layout()
plt.show()
```

6.3.4 Figure 2

```
# Figure 2: Comparing different parameter initializations
# Overlay specific iterations of parameter distributions
n = 1000
gamma = 2.3

# Generate power-law distributed degree sequence
degrees = generate_power_law_degrees(n, gamma)
edges = configuration_model(degrees)
A = convert_to_adjacency_matrix(edges, n)

initializations = [
    ("Random Initialization", np.random.rand(n)),
    ("Power Law Initialization", np.random.zipf(2.3, size=n)/4.3),
    ("Normal Initialization SD 0.1", np.random.normal(0.5, 0.1, n)),
    ("Normal Initialization SD 0.5", np.random.normal(0.5, 0.5, n))
]

# Clip the parameters and print the number of parameters at 0 and 1
initializations = [
```

```
    (title, np.clip(parameters, 0, 1)) for title, parameters in initializations
]

iterations_to_plot = [0, 50, 100]  # Iterations 0, 50, and 100 (0-indexed)
bins = np.linspace(0, 1, 100)

# Set up the plot with 4 subfigures
fig, axs = plt.subplots(2, 2, figsize=(15, 12))
axs = axs.flatten()

for ax, (title, parameters) in zip(axs, initializations):
    for i in range(101):
        if i in iterations_to_plot:
            # Overlay parameter distribution
            ax.hist(parameters, bins=bins, density=True, alpha=0.5, label=f'Iteration {i}')
        scores = calculate_scores_equal(parameters)
        parameters = update_parameters_linear(scores, parameters, A)



    # Add titles and labels
    ax.set_title(title)
    ax.set_xlabel("Parameter Value")
    ax.set_ylabel("Density")
    ax.legend()

plt.tight_layout()
plt.show()
```

6.3.5 Figure 3

```
n = 1000
gamma = 2.3

degrees = generate_power_law_degrees(n, gamma)
edges = configuration_model(degrees)
A = convert_to_adjacency_matrix(edges, n)

# Parameters
num_generations = 100
num_runs = 1000
noise_levels = [0, 0.25, 0.5, 0.75]


all_scores = {noise: np.zeros((num_runs, num_generations, n)) for noise in noise_levels}

# Repeat the operation for num_runs times
for run in tqdm(range(num_runs)):
    for noise in noise_levels:
        # Initialize with the same distribution for all noise levels
        parameters = np.random.normal(0.5, 0.3, n)
        for i in range(num_generations):
            scores = calculate_scores_equal_noisy(parameters, noise)
```

```
            parameters = update_parameters_linear(scores, parameters, A)
            all_scores[noise][run, i, :] = scores

# Calculate the percentiles
percentiles = {noise: np.zeros((num_generations, 3)) for noise in noise_levels}
for noise in noise_levels:
    for i in range(num_generations):
        percentiles[noise][i, 0] = np.percentile(all_scores[noise][:, i, :], 10)
        percentiles[noise][i, 1] = np.percentile(all_scores[noise][:, i, :], 50)
        percentiles[noise][i, 2] = np.percentile(all_scores[noise][:, i, :], 90)

# Plotting the percentiles over generations for different noise levels
fig, axes = plt.subplots(2, 2, figsize=(14, 10), sharex=True, sharey=True)
axes = axes.flatten()

for idx, noise in enumerate(noise_levels):
    axes[idx].plot(range(num_generations), percentiles[noise][:, 0], label='10th percentile')
    axes[idx].plot(range(num_generations), percentiles[noise][:, 1], label='50th percentile')
    axes[idx].plot(range(num_generations), percentiles[noise][:, 2], label='90th percentile')
    axes[idx].set_title(f'Noise level: {noise}')
    axes[idx].set_xlabel('Generations')
    axes[idx].set_ylabel('Scores')
    axes[idx].grid(True)
    axes[idx].legend()

plt.tight_layout()
plt.show()
```

6.3.6 Referral Mechanism

```
    def get_highest_scoring_neighbor(adj_matrix, node_scores, node):
    neighbors = np.where(adj_matrix[node] == 1)[0]
    if len(neighbors) == 0:
        #print("No neighbors")
        return None
    highest_neighbor = neighbors[np.argmax([node_scores[n] for n in neighbors])]
    return highest_neighbor

def reshuffle_matrix(adj_matrix, node_scores, referral_rate, min_degree=0):
    num_nodes = len(adj_matrix)
    new_adj_matrix = np.copy(adj_matrix)
    reshuffled_nodes = []
    for node in range(num_nodes):
        if random.random() <= referral_rate:
            highest_neighbor = get_highest_scoring_neighbor(adj_matrix, node_scores, node)

            neighbors = np.where(adj_matrix[node] == 1)[0]
            if len(neighbors) == 0:
                continue

            neighbors_of_neighbor = neighbors[neighbors != highest_neighbor]
            if len(neighbors_of_neighbor) == 0:
                continue
            random_neighbor = random.choice(neighbors_of_neighbor)
```

```
            random_neighbor_neighbors = np.where(adj_matrix[random_neighbor] == 1)[0]

            if len(random_neighbor_neighbors) <= 1:
                continue


            remove_candidate = random.choice(random_neighbor_neighbors[random_neighbor_neighbors
            != node])

            if len(np.where(adj_matrix[remove_candidate] == 1)[0]) <= min_degree:
                continue

            if new_adj_matrix[random_neighbor, remove_candidate] == 1
            and new_adj_matrix[random_neighbor, highest_neighbor] == 0:
                if remove_candidate != highest_neighbor:
                    new_adj_matrix[random_neighbor, remove_candidate] = 0
                    new_adj_matrix[remove_candidate, random_neighbor] = 0

                    new_adj_matrix[random_neighbor, highest_neighbor] = 1
                    new_adj_matrix[highest_neighbor, random_neighbor] = 1

                    reshuffled_nodes.append([random_neighbor, remove_candidate, highest_neighbor])


    return new_adj_matrix, reshuffled_nodes
```

6.3.7 Figure 4

```
n = 5000
gamma = 2.3

degrees = generate_power_law_degrees(n, gamma)
edges = configuration_model(degrees)
A = convert_to_adjacency_matrix(edges, n)

# Parameters
# We take 100 gens here as for a network of this size,
# past 100 generations the scores are already converging and don't show much interesting behaviour
num_generations = 100
num_runs = 50
referral_rates = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

def run_simulation_referral_percentiles(A, num_generations, num_runs, referral_rates,
min_degree=0, noise=0):
    # Initialize lists to store average scores and percentiles
    average_scores_all_runs = {rate: np.zeros(num_generations) for rate in referral_rates}
    percentile_scores_all_runs = {rate: {10: np.zeros(num_generations), 50: np.zeros(num_generations), 9
    all_scores_per_gen = {rate: [[] for _ in range(num_generations)] for rate in referral_rates}

    # Repeat the operation for num_runs times
    for run in tqdm(range(num_runs)):
        base_parameters = np.random.normal(0.5, 0.3, n)
        for rate in referral_rates:
```

```
            parameters = base_parameters.copy()
            adjacency_matrix = A.copy()
            for i in range(num_generations):
                if noise == 0:
                    scores = calculate_scores_equal(parameters)
                else:
                    scores = calculate_scores_equal_noisy(parameters, noise)
                parameters = update_parameters_linear(scores, parameters, adjacency_matrix)

                adjacency_matrix, _ = reshuffle_matrix(adjacency_matrix, scores, referral_rate=rate,
                min_degree=min_degree)

                average_scores_all_runs[rate][i] += np.mean(scores)
                all_scores_per_gen[rate][i].extend(scores)

    # Calculate the average scores and percentiles over all runs
    for rate in referral_rates:
        average_scores_all_runs[rate] /= num_runs
        for i in range(num_generations):
            if len(all_scores_per_gen[rate][i]) > 0:
                percentile_scores_all_runs[rate][10][i] = np.percentile(all_scores_per_gen[rate][i],
                10)
                percentile_scores_all_runs[rate][50][i] = np.percentile(all_scores_per_gen[rate][i],
                50)
                percentile_scores_all_runs[rate][90][i] = np.percentile(all_scores_per_gen[rate][i],
                90)
            else:
                percentile_scores_all_runs[rate][10][i] = np.nan
                percentile_scores_all_runs[rate][50][i] = np.nan
                percentile_scores_all_runs[rate][90][i] = np.nan

    return average_scores_all_runs, percentile_scores_all_runs

average_scores, percentile_scores = run_simulation_referral_percentiles(A, num_generations,
num_runs, referral_rates)

# Select the referral rates to plot
selected_rates = [0.1, 0.3, 0.6, 0.9]
colors = {0.1: 'b', 0.3: 'g', 0.6: 'm', 0.9: 'r'}  # Define colors for each referral rate

fig, axes = plt.subplots(2, 2, figsize=(15, 10))

for i, rate in enumerate(selected_rates):
    ax = axes[i // 2, i % 2]
    generations = range(num_generations)

    # Prepare scores for plotting on log-log scale
    adjusted_10th_percentile = percentile_scores[rate][10]
    adjusted_50th_percentile = percentile_scores[rate][50]
    adjusted_90th_percentile = percentile_scores[rate][90]

    ax.plot(generations, adjusted_50th_percentile, label=f'50th Percentile', color=colors[rate])
    ax.fill_between(generations,
                    adjusted_10th_percentile,
```

```
                    adjusted_90th_percentile,
                    color=colors[rate],
                    alpha=0.2)
    ax.plot(generations, adjusted_10th_percentile, linestyle='--',
    color=colors[rate], alpha=0.7, label=f'10th Percentile')
    ax.plot(generations, adjusted_90th_percentile, linestyle='--',
    color=colors[rate], alpha=0.7, label=f'90th Percentile')

    ax.set_xlabel('Generations')
    ax.set_ylabel('Scores')
    # log scale on both axes
    ax.set_xscale('log')
    ax.set_yscale('log')
    ax.set_title(f'Scores Distribution Over Generations (Referral rate: {rate})')
    ax.legend()
    ax.grid(True, which="both", ls="--")

plt.tight_layout()
plt.show()
```

6.3.8 Figure 5

```
n = 500
gamma = 2.3

degrees = generate_power_law_degrees(n, gamma)
edges = configuration_model(degrees)
A = convert_to_adjacency_matrix(edges, n)

num_generations = 200
num_runs = 100
referral_rates = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

def run_simulation_referral_score_ref_rate_filter(A, num_generations, num_runs,
referral_rates, min_degree=0, noise=0):
    # Initialize lists to store average scores at different points
    # Empirical points around where we hit the thresholds of 0.6, 0.7, 0.8, 0.9 for this network
    points = [15, 35, 100, 150]
    average_scores_all_runs = {rate: np.zeros((4,)) for rate in referral_rates}

    # Repeat the operation for num_runs times
    for run in tqdm(range(num_runs)):
        base_parameters = np.random.normal(0.5, 0.2, len(A))
        for rate in referral_rates:
            parameters = base_parameters.copy()
            adjacency_matrix = A.copy()
            for i in range(num_generations):
                if noise == 0:
                    scores = calculate_scores_equal(parameters)
                else:
                    scores = calculate_scores_equal_noisy(parameters, noise)
                parameters = update_parameters_linear(scores, parameters, adjacency_matrix)
                adjacency_matrix, _ = reshuffle_matrix(adjacency_matrix, scores, rate, min_degree)
```

```
                if i in points:
                    index = points.index(i)
                    average_scores_all_runs[rate][index] += np.mean(scores)

    # Calculate the average scores over all runs
    for rate in referral_rates:
        average_scores_all_runs[rate] /= num_runs

    return average_scores_all_runs

average_score_trends = run_simulation_referral_score_ref_rate_filter(A, num_generations,
num_runs, referral_rates)

points_labels = ['15', '35', '100', '150']

# Create a figure with 4 subplots
fig, axs = plt.subplots(2, 2, figsize=(12, 10))
axs = axs.flatten()

# Plotting the data
for idx, point_label in enumerate(points_labels):
    axs[idx].plot(referral_rates,
    [average_score_trends[rate][idx] for rate in referral_rates], marker='o')
    axs[idx].set_title(f'Average Score at {point_label} Generations')
    axs[idx].set_xlabel('Referral Rate')
    axs[idx].set_ylabel('Average Score')
    axs[idx].grid(True)

plt.tight_layout()
plt.show()
```

6.3.9 Figure 6

```
n = 500
gamma = 2.3

degrees = generate_power_law_degrees(n, gamma)
edges = configuration_model(degrees)
A = convert_to_adjacency_matrix(edges, n)

num_generations = 200
num_runs = 1
referral_rates = [0.1, 0.9]
def run_simulation_referral_distribution_slice(A, num_generations, num_runs,
referral_rates, sd, min_degree=0, noise=0):
    checkpoints = [15, 35, 100, 150]
    score_distribution = {rate: {checkpoint: np.zeros(n) for
    checkpoint in checkpoints} for rate in referral_rates}

    # Repeat the operation for num_runs times
    for run in tqdm(range(num_runs)):
        base_parameters = np.random.normal(0.5, sd, n)
        for rate in referral_rates:
            parameters = base_parameters.copy()
```

```
            adjacency_matrix = A.copy()
            # This ensures that the highest scoring node is always the same
            highest_scoring_index = np.argmax(parameters)
            for i in range(num_generations):
                if noise == 0:
                    scores = calculate_scores_equal(parameters)
                else:
                    scores = calculate_scores_equal_noisy(parameters, noise)
                parameters = update_parameters_linear(scores, parameters, adjacency_matrix)

                adjacency_matrix, reshuffled_nodes = reshuffle_matrix(adjacency_matrix, scores,
                referral_rate=rate)

                # Store score distribution at the checkpoints
                if i in checkpoints and run == 0:
                    score_distribution[rate][i] = parameters.copy()

    return score_distribution

score_distribution_slice = run_simulation_referral_distribution_slice(A, num_generations,
num_runs, referral_rates, 0.2)

# Plot histogram slices of scores of 0.1 and 0.9 overlaid at
# 4 different time points 25%, 50%, 75%, 100%

fig, axes = plt.subplots(2, 2, figsize=(15, 10))

for i, checkpoint in enumerate([15, 35, 100, 150]):
    ax = axes[i // 2, i % 2]
    for rate in [0.1, 0.9]:
        ax.hist(score_distribution_slice[rate][checkpoint], bins=50, alpha=0.5,
        label=f'Referral rate: {rate}', density=True)
    ax.set_title(f'Score Distribution at Generation {checkpoint}')
    ax.set_xlabel('Scores')
    ax.set_ylabel('Density')
    ax.legend()
    ax.grid(True)

plt.tight_layout()
plt.show()
```