Theory and Application of Signed Spectral Clustering

Philippe Sawaya
psawaya@seas.upenn.edu

João Sedoc
joao@seas.upenn.edu

Abstract

In this paper, we describe a step-by-step implementation of the signed spectral clustering algorithm and the application of signed spectral clustering to recipe rating prediction. In particular, using normalized pointwise mutual information (NPMI) as a pairwise affinity metric, we present a new approach to recipe rating prediction that uses ingredient clusters as features for rating prediction.

1 Spectral Graph Theory Overview

1.1 Undirected and Directed Graphs

We define a directed graph G as a tuple (V, E) where:

1. $V = \{1, 2, ..., n\}$ is a set of n *vertices*.
2. $E \subseteq V \times V$ is a set of *edges* comprising ordered pairs of distinct vertices (u, v) where $u, v \in V$ and $u \neq v$.

An undirected graph G, then, is a tuple (V, E) where:

1. $V = \{1, 2, ..., n\}$ is a set of n *vertices*.
2. $E \subseteq V \times V$ is a set of *edges* comprising ordered pairs of distinct vertices where $u, v \in V$, $u \neq v$, and $(u, v) \in E$ if and only if $(v, u) \in E$.

1.2 Weighted Graphs

A weighted graph G is a tuple (V, W) where:

1. $V = \{1, 2, ..., n\}$ is a set of n *vertices*.
2. W is an $n \times n$ symmetric *weight matrix* where $W_{ij} \geq 0$ $(\forall i, j \in V)$ and $W_{ii} = 0$ $(\forall i \in V)$. The underlying undirected graph $G(V, E)$ contains all edges $\{(u, v) \mid W_{uv} > 0\}$.

1.3 Signed Weighted Graphs

A signed weighted graph G is a tuple (V, W) where:

1. $V = \{1, 2, ..., n\}$ is a set of n *vertices*.
2. W is an $n \times n$ symmetric *weight matrix* where $W_{ii} = 0$ $(\forall i \in V)$. The underlying undirected graph $G(V, E)$ contains all edges $\{(u, v) \mid W_{uv} \neq 0\}$.

Note that this definition is simply that of a weighted graph except we have relaxed the requirement that $W_{ij} \geq 0$ $(\forall i, j \in V)$ thereby allowing both positive and negative weights.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

1.4 Degree Matrix

Given a weighted graph $G(V, W)$, for any vertex $i \in V$ define the *degree* of i, $d(i)$, as:

$$d(i) = \sum_{j \in V,\, j \neq i} W_{ij}$$

Then, the degree matrix $D(W)$ of W is the diagonal matrix $D(W) = \text{diag}(d(1), ..., d(n))$.

2 Signed Spectral Clustering Implementation

2.1 Optimization Problem

The signed spectral clustering optimization problem is stated as:

$$\begin{aligned} &\text{Minimize:} && \sum_{j=1}^{k} \frac{(X^j)^T \overline{L} X^j}{(X^j)^T \overline{D} X^j} \\ &\text{Subject to:} && (X^i)^T \overline{D} X^j = 0 \ (\forall 1 \leq i \neq j \leq k) \end{aligned}$$

However, we can formulate a relaxed version of the problem:

$$\begin{aligned} &\text{Minimize:} && tr(Y^T \overline{D}^{-\frac{1}{2}} \overline{L} \overline{D}^{-\frac{1}{2}} Y) \\ &\text{Subject to:} && Y^T Y = I \end{aligned}$$

Let us denote the k clusters into which we wish to partition V as $C_1, C_2, ..., C_k$. Then, we define the solution X for all $i \in V$, $j \in [1, k]$ as:

$$X_{ij} = \begin{cases} 1 & i \in C_j \\ 0 & otherwise \end{cases}$$

2.2 Inputs and Outputs

The signed spectral clustering algorithm takes as input the 6-tuple $< W, K, \epsilon, n_{iter}, T >$ where:

1. W is an $n \times n$ symmetric *signed weight matrix* where $W_{ii} = 0 \ (\forall i \in V)$. The underlying undirected graph $G(V, E)$ contains all edges $\{(u, v) \mid W_{uv} \neq 0\}$.
2. $k \in \mathbb{Z}^+$ denotes the number of clusters to partition V into.
3. $\epsilon \in \mathbb{R}^+$ is a terminal threshold.
4. $n_{iter} \in \mathbb{Z}^+$ denotes the maximum number of iterations to refine the partition of V into k clusters.
5. $T \in \{$*"Hard", "Soft", "Flexible"*$\}$ denotes the type of clustering to apply.

At the termination of the algorithm, the tuple $< K, X^* >$ is returned where:

1. K is an $n \times 1$ matrix where K_i denotes the cluster to which vertex $i \in V$ is assigned. Note that $K_i \in [1, k] \ (\forall i \in V)$.
2. X^* is an $n \times k$ matrix that is a solution to the above optimization problem.

2.3 Implementation Details

This implementation of the signed spectral clustering algorithm uses Python 3.6 and Numpy v1.14.

2.4 Algorithm

2.4.1 Computing the Degree Matrix & Laplacians

The algorithm first computes the signed degree matrix $\overline{D}$, the signed Laplacian $\overline{L}$, and the signed normalized Laplacian $\overline{L}_{sym}$. Let $\overline{d_i}$ denote the signed degree of a vertex $i \in V$. Then, we define $\overline{d_i}$ for each $i \in V$ as:

$$\overline{d_i} = \sum_{j=1}^{n} |w_{ij}|$$

The signed degree matrix is then the $n \times n$ diagonal matrix $\overline{D}$ where for all $i, j \in [1, n]$:

$$\overline{D_{ij}} = \begin{cases} \overline{d_i} & i \neq j \\ 0 & otherwise \end{cases}$$

We then define the signed Laplacian $\overline{L}$ and signed normalized Laplacian $\overline{L}_{sym}$ as:

$$\overline{L} = \overline{D} - W$$
$$\overline{L}_{sym} = \overline{D}^{-\frac{1}{2}} \overline{L} \overline{D}^{-\frac{1}{2}} = I - \overline{D}^{-\frac{1}{2}} W \overline{D}^{-\frac{1}{2}}$$

The code below represents the above computations where `deg_sums` is the $n \times 1$ vector $\{\overline{d_1}, \overline{d_2}, ..., \overline{d_n}\}$, `deg_mat` = $\overline{D}$, `deg_inv` = $\overline{D}^{-\frac{1}{2}}$, `lap` = $\overline{L}$, and `lap_sym` = $\overline{L}_{sym}$.

```
deg_sums = np.absolute(w).sum(axis=0)
deg_mat = np.diagflat(deg_sums)
deg_sums_sqrt = np.sqrt(deg_sums)
deg_sums_sqrt[deg_sums_sqrt == 0] = 1e-14
deg_inv = np.diagflat(np.reciprocal(deg_sums_sqrt))
lap = deg_mat - w
lap_sym = deg_inv.dot(lap).dot(deg_inv)
```

2.4.2 Computing the Solution to the Relaxed Problem

The algorithm then computes a solution to the relaxed problem as defined in section 2.1. Initially, we let $Z = \overline{D}^{-\frac{1}{2}} U$ where U is a $n \times k$ matrix containing the k smallest eigenvalues of $\overline{L}_{sym}$. Note that Z is a minimum of the relaxed problem. Since it is much more computationally efficient to compute the largest eigenvalues of a symmetric matrix than the smallest, rather than directly compute the k smallest eigenvalues of $\overline{L}_{sym}$, we compute the k largest eigenvalues of $2I - \overline{L}_{sym}$.

The below code reflects the computation of the eigenvectors of $\overline{L}_{sym}$ where `u_k` = U and `z1` = Z:

```
u, _, _ = np.linalg.svd(2 * np.eye(n_vert) - lap_sym)
u_k = u[:, 0:n_clusters]
z1 = deg_inv.dot(u_k)
```

2.4.3 Procedure to Compute a Discrete Solution

Recall that Z is a solution to the relaxed problem. As described in Gallier [4], ZQ is a solution to the original optimization problem for all $k \times k$ matrices Q with non-zero and pairwise orthogonal columns. If we denote the discrete solution X, then the desired solution is that with minimal $\|X - ZQ\|_F^2$, the Froebenius norm. Let $\phi(X, Q) = \|X - ZQ\|_F^2$ as in Gallier [4]. It follows that we must identify (X, Q) pairs such that $\phi(X, Q)$ is minimized.

The difficulty of this optimization problem necessitates the following approach to minimizing $\phi(X, Q)$: the algorithm first minimizes $\phi(X, Q)$ holding X constant and then minimizes $\phi(X, Q)$ holding Q constant.

Gallier [4] then further decomposes the problem by letting $Q = R\Lambda$ where Λ represents a diagonal invertible scaling matrix and $R \in O(k)$ where $O(k)$ denotes the orthogonal group in dimension k. Let us denote $\phi(X, Z, R, \Lambda) = \|X - ZR\Lambda\|_F^2$. Therefore, the aforementioned two-stage approach becomes the following three-stage approach:

1. Minimize $\phi(X, Z, R, \Lambda)$ by holding Z, R, Λ fixed
2. Minimize $\phi(X, Z, R, \Lambda)$ by holding X, Z, Λ fixed
3. Minimize $\phi(X, Z, R, \Lambda)$ by holding X, Z, R fixed

We implemented a helper function for the calculation of $\phi(X, Z, R, I) = \|X - ZR\|_F^2$:

```
def norm_diff(X, Z, R):
    Q = Z.dot(R)
    return np.sqrt(np.trace((X - Q).H.dot(X - Q)))
```

If we let $\phi_j = \phi(X, Z, R, \Lambda)$ using the values of X, Z, R, Λ computed on iteration $\gamma \in [1, n_{iter}]$, then our algorithm will iteratively perform the above three steps for a maximum of n_{iter} iterations or until $\phi_\gamma - \phi_{\gamma-1} < \epsilon$.

2.4.4 Minimizing $\phi(X, Z, R, \Lambda)$ with Fixed Z, R, Λ

The first stage itself requires a number of steps to compute the X that minimizes $\phi(X, Z, R, \Lambda)$. To compute X we must first compute $\widehat{X}$, the shape of X, but to compute $\widehat{X}$ we must first compute $\overline{X}$, an $n \times k$ matrix such that each row i has one non-zero entry. $\overline{X}$ represents the unnormalized shape of X. Let $Y = ZR$. Then, as in Gallier [4], let J be an $n \times 1$ vector where for all $i \in [1, n]$, J_i is defined as:

$$J_i = \{j \in \{1, 2, ..., k\} \mid Y_{ij} = \max_{1 \leq j \leq k} Y_{ij}\}$$

Now, for each row i of $\overline{X}$, let the single non-zero entry on this row be at column J_i and so we can now define $\overline{X}_{ij}$ for all $i \in [1, n], j \in [1, k]$ as:

$$\overline{X_{ij}} = \begin{cases} 1 & j = J_i \\ 0 & otherwise \end{cases}$$

In the below code, `zr1` = Y, `x1` = $\overline{X}$, and `J` = J.

```
zr1 = z.dot(r)
k = zr1.shape[1]
zr2, _ = flip_negative_columns(zr1)
    for st in range(0, 2):
        zr = zr1 if st == 0 else zr2
        n = zr.shape[0]
        x1 = np.zeros((n, k))
        J = np.argmax(zr.H, axis=0)
        for i in range(0, n):
            x1[i, J[0, i]] = 1
```

However, Gallier [4] notes that for any Y with columns of negative averages, an alternate X' can be derived by computing a Y' such that if the average of a column j in Y is negative, then $Y'^j = -Y^j$, ensuring that that same column in Y' has a non-negative average. Note that `zr2` in the above code corresponds to Y' and `flip_negative_columns(zr1)` is the sub-routine that performs this non-negatizing of Y:

```
def flip_negative_columns(z):
    n = z.shape[0]
    k = z.shape[1]
    rr = np.eye(k)
    col_sum = np.ones((1, n)).dot(z)
    z2 = z
    for i in range(0, k):
        if col_sum[0, i] < 0:
            z2[:, i] = -z[:, i]
```

```
            rr[i, i] = -1
    return [z2, rr]
```

After $\overline{X}$ has been computed, it remains to compute $\widehat{X}$ and X itself. Gallier [4] notes a potential issue: that $\overline{X}$ may have columns with sums of zero. Therefore, to compute $\widehat{X}$ from $\overline{X}$, we simply identify columns with at least one non-zero value and shift those values to columns with no non-zero values until every column has a non-zero sum.

The exact procedure is as follows: repeatedly identify indices j such that column j of $\overline{X}$ has a sum of zero, identify the smallest j' such that column j' of $\overline{X}$ has the maximum sum of any column, and then identify the smallest row index i such that $\overline{X_{ij'}} = 1$. Finally, we let $\overline{X_{ij'}} = 0$ and $\overline{X_{ij}} = 0$ thereby shifting the value 1 to column j ensuring that j has a non-zero sum. We implement this procedure with the below code where `sum_idx` denotes the column j' with the maximum sum and `row_idx[j]` denotes contains the row index of the first non-zero entry in column j. At the end of this procedure, `x1` is $\widehat{X}$.

```
x1_sum = np.sum(x1, axis=0)
row_idx = np.argmax(x1, axis=0)
sum_idx = np.argmax(x1_sum, axis=0)
for j in range(0, int(k)):
    if x1_sum[j] == 0:
        x1[row_idx[sum_idx]][sum_idx] = 0
        x1[row_idx[sum_idx]][j] = 1
        x1_sum = sum(x1)
        row_idx = np.argmax(x1, axis=0)
        sum_idx = np.argmax(x1_sum, axis=0)
```

Finally, to compute X from $\widehat{X}$, we simply normalize $\widehat{X}$ such that $\|X\|_F = \|Z\|_F$. In the below code, `xx1` denotes the X computed from Y and `xx2` denotes the X' computed from Y'.

```
if st == 0:
    xx1 = a*x1
else:
    xx2 = a*x1
```

The overall procedure for computing the X that minimizes $\phi(X, Z, R, \Lambda)$ is below. Note the values `n1` and `n2` which respectively denote $\phi(X, Z, R, \Lambda)$ and $\phi(X', Z, R, \Lambda)$. This sub-routine then returns X if $\phi(X, Z, R, \Lambda) < \phi(X', Z, R, \Lambda)$ and returns X' otherwise.

```
def find_hard_clusters(z, r, a):
    zr1 = z.dot(r)
    k = zr1.shape[1]
    zr2, rn = flip_negative_columns(zr1)
    for st in range(0, 2):
        zr = zr1 if st == 0 else zr2
        n = zr.shape[0]
        x1 = np.zeros((n, k))
        J = np.argmax(zr.H, axis=0)
        for i in range(0, n):
            x1[i, J[0, i]] = 1

        x1_sum = np.sum(x1, axis=0)
        row_idx = np.argmax(x1, axis=0)
        sum_idx = np.argmax(x1_sum, axis=0)
        for j in range(0, int(k)):
            if x1_sum[j] == 0:
                x1[row_idx[sum_idx]][sum_idx] = 0
                x1[row_idx[sum_idx]][j] = 1
```

```
                x1_sum = sum(x1)
                row_idx = np.argmax(x1, axis=0)
                sum_idx = np.argmax(x1_sum, axis=0)
            if st == 0:
                xx1 = a*x1
            else:
                xx2 = a*x1

        n1 = norm_diff(xx1, zr1, np.eye(zr1.shape[1]))
        n2 = norm_diff(xx2, zr2, np.eye(zr2.shape[1]))
        x = xx2 if n1 > n2 else xx1
        return [x, rn]
```

2.4.5 Minimizing $\phi(X, Z, R, \Lambda)$ with Fixed Z, X, Λ

In this stage, we seek to find an $R \in O(k)$ that minimizes $\phi(X, Z, R, \Lambda)$. We first let $\Lambda = I$ and so we seek to minimize the quantity $\|X - ZR\|_F^2$. The algorithm computes R by letting $R = UV^T$ where $U\Sigma V^T$ is a singular value decomposition of $Z^T X$:

```
def find_r(x, z):
    u, _, v = np.linalg.svd(z.H.dot(x))
    return u.dot(v.H)
```

2.4.6 Minimizing $\phi(X, Z, R, \Lambda)$ with Fixed Z, X, R

The final stage aims to compute the invertible diagonal scaling matrix Λ that minimizes $\phi(X, Z, R, \Lambda)$. As Gallier [4] describes, given two fixed $n \times k$ matrices X, Z, the unique Λ that minimizes $\phi(X, Z, R, \Lambda)$ is $\Lambda = \text{diag}(\lambda_1, ..., \lambda_k)$ where for all $j \in [1, k]$, λ_j is defined as:

$$\lambda_j = \frac{(Z^T X)_{jj}}{\|Z_j\|_2^2}$$

We compute Λ using the below function where `y` = $Z^T X$ and `np.lingalg.inv(z.h.dot(z))` computes the pseudo-inverse $(Z^T Z)^{-1}$:

```
def find_a(x, z):
    y = z.H.dot(x)
    return np.lingalg.inv(z.h.dot(z)).dot(y)
```

2.4.7 Finding "Good" Initial X, Z, and R

Before iteratively repeating the above three steps to minimize $\phi(X, Z, R, \Lambda)$, the algorithm must first determine the initial values of X, Z, R, and Λ. Denote these values X_0, Z_0, R_0, and Λ_0 and let $\Lambda_0 = I$. Recall from section 2.3.2 that we have an initial value of Z that was the solution to the relaxed problem. Denote this Z as Z_1.

We will also consider an alternative Z_0 that we will denote Z_2. Let R_1 be a $k \times k$ symmetric matrix such that $Z_1^T Z_1$ can be diagonalized as $Z_1^T Z_1 = R_1 \Sigma R_1^T$. Then, $Z_2 = Z_1 R_1$:

```
_, r1 = np.linalg.eig(z1.H.dot(z1))
z2 = z1.dot(r1)
```

Now, consider the sub-routine described in 2.3.4 to compute an X that minimizes $\phi(X, Z, R, \Lambda)$ given fixed Z, R, Λ. Note that we have identified two potential values for Z_0 and $\Lambda_0 = I$. Therefore, if we can identify a suitable R_0, then we can compute an X_1 that minimizes $\phi_1 = \phi(X_1, Z_1, R_0, I)$ and an X_2 that minimizes $\phi_2 = \phi(X_2, Z_2, R_0, I)$ and then let $X_0 = X_1$ if $\phi_1 < \phi_2$ and X_2 otherwise.

Gallier [4] describes a method for computing an initial R that entails selecting the k most orthogonal rows from Z with respect to each other. We implement this method with the below function:

```
def pick_orthogonal_rows(Z):
    zeroes = np.sum(np.abs(Z.H), axis=0)
    Z = Z[zeroes.nonzero()[1], :]
    n = Z.shape[0]
    k = Z.shape[1]
    R = np.eye(k)
    if n < k:
        return R

    Z = np.delete(Z, 0, axis=0)
    c = np.zeros((n-1, 1))
    for i in range(1, k):
        # Find index of row in Z most orthogonal to column i of R
        c += np.abs(Z.dot(R[:, i-1])).T
        min_c = np.argmin(c)

        # Add this row to R2
        R[:, i] = Z[min_c, :].conj()

        # Delete the row from Z
        Z = np.delete(Z, min_c, 0)
        c = np.delete(c, min_c, 0)

    nrc = np.zeros((k, k))
    rr = R.conj().T.dot(R)
    for i in range(0, k):
        nrc[i, i] = np.reciprocal(np.sqrt(rr[i, i]))
    return R.dot(nrc.conj().T)
```

Finally, to compute the X_0, we compute four candidate X_1, X_2, X_3, X_4 with corresponding $\phi_1, \phi_2, \phi_3, \phi_4$ and let $X_0 = X_i$ where $i = \text{argmin}_j\{\phi_j\}$. To compute each X_i, we use the subroutine from 2.3.4 that computes an X that minimizes $\phi(X, Z, R, \Lambda)$ given fixed Z, R, Λ and we the following four Z, R pairs:

1. Z_1 and $R = I$
2. Z_1 and $R = R2a$
3. Z_2 and $R = I$
4. Z_2 and $R = R2b$

Note that $R2a$ is the R computed by the above `pick_orthogonal_rows` procedure given $Z1$ as input and $R2b$ is the R computed by the procedure given $Z2$ as input. We finally select the candidate X that minimizes $\phi(X, Z, R, \Lambda)$. In the below code, `r2a` = $R2a$, `r2b` = $R2b$, `n1` = ϕ_1, and `xc1` = X_1. The remaining X_2, X_3, X_4 and ϕ_2, ϕ_3, ϕ_4 follow this convention.

```
def compute_initial_x_z_r(Z1, Z2, k, a, f):
    R2a = pick_orthogonal_rows(Z1)
    R2b = pick_orthogonal_rows(Z2)

    X1, RR1 = f(Z1, np.eye(k), a)
    R1 = RR1
    n1 = norm_diff(X1, Z1, R1)

    X2, RR2 = f(Z1, R2a, a)
    R2 = R2a.dot(RR2)
    n2 = norm_diff(X2, Z1, R2)
```

```
    X3, RR3 = f(Z2, np.eye(k), a)
    R3 = RR3
    n3 = norm_diff(X3, Z2, R3)

    X4, RR4 = f(Z2, R2b, a)
    R4 = R2b.dot(RR4)
    n4 = norm_diff(X4, Z2, R4)

    norms = [n1, n2, n3, n4]
    x_candidates = [X1, X2, X3, X4]
    z_candidates = [Z1, Z1, Z2, Z2]
    r_candidates = [R1, R2, R3, R4]
    X0 = x_candidates[np.argmin(norms)]
    Z0 = z_candidates[np.argmin(norms)]
    R0 = r_candidates[np.argmin(norms)]
    return [X0, Z0, R0]
```

Having determined a "good" enough R_0 using the above procedure, since we now have values for X_0 and Z_0, we can use the procedure in 2.3.5 to find an R that minimizes $\phi(X_0, Z_0, R, I)$:

```
rc, z, xc = compute_initial_x_z_r(z1, z2, n_clusters, a, f)
rc = find_r(xc, z)
```

2.4.8 Computing a Discrete Solution

Having determined an iterative three-stage procedure to identify a discrete solution X^* that minimizes $\phi(X^*, Z, R, \Lambda)$ and having identified "good" initial X_0, Z_0, R_0, Λ_0, we can finally repeat the three-stage procedure to convergence (or until we reach a termination condition defined by ϵ and n_{iter}:

```
ern = norm_diff(X0, Z, R0)
er = ern + 1

X_curr = X0
R_curr = R0

for iteration in range(max_iters):
    if ern >= er:
        break

    # Find best Xc given fixed Z, R, Lambda
    X_curr, _ = f(Z, R_curr, a)
    diff_xc = (X_curr - X0)/a
    ndxc = np.trace(diff_xc.conj().T.dot(diff_xc))/2.

    # Halt if threshold is passed
    if ndxc < threshold:
        ern = er
    else:
        # Find best R given fixed Xc, Z
        R_curr = find_r(X_curr, Z)
        er = ern

        # Compute phi(Xc, Z, Rc) with new Rc
        ern = norm_diff(X_curr, Z, R_curr)

        # If (X0, Z, R0) < phi(X', Z, R'), then maintain X = X0, R = R0
        if er < ern:
            X_curr = X0
            R_curr = R0
```

```
        # Otherwise, we have found a better estimate
        else:
            X0 = X_curr
            R0 = R_curr
```

Finally, we re-normalize our estimate estimate X_f to yield X^* and identify the clusters to which each $i \in V$ belongs by letting:

$$K_{ij} = \begin{cases} 1 & j = \text{argmax}_{j' \in [1,k]} X^*[j'] \\ 0 & otherwise \end{cases}$$

This procedure is reflected in the below code where `XX` is X^*,`X_curr` is X_f, and `indices` is K.

```
XX = (1./a) * X_curr
indices = np.argmax(XX.conj().T, axis=0)
```

2.4.9 Computing a Probabilistic Clusters: "Soft Clustering"

The above procedure computes "hard" clusters, assigning vertices to singular clusters. We also implemented a "soft clustering" algorithm that simply uses a different procedure to minimize $\phi(X, Z, R, \Lambda)$ with fixed Z, R, Λ. The soft clustering algorithm then produces a solution X^* such that X^* is an $n \times k$ matrix where $X^*_{ij} = \Pr(K_i = j)$ for all $i \in V,\ j \in [1, k]$.

Similarly to the hard clustering procedure in section 2.4.4, we first compute Y and Y' where $Y = ZR$ and Y' is Y except the sign of columns with negative averages has been flipped. This ensures that every column in Y' has non-negative average.

Then, rather than perform the "1-shifting" procedure from section 2.4.4 to produce $\widehat{X}$, we compute two candidate $\widehat{X}_1$ and $\widehat{X}_2$ such that for every column $j \in [1, k]$:

$$\beta_1^j = \sum_{i \in [1,n]} Y_{ij}$$

$$\beta_2^j = \sum_{i \in [1,n]} Y'_{ij}$$

$$\widehat{X}_1^j = \frac{Y^j}{\beta_1^j}$$

$$\widehat{X}_2^j = \frac{Y'^j}{\beta_2^j}$$

Finally, as in the hard clustering procedure, we obtain the two candidate solutions X_1 and X_2 by normalizing $\widehat{X}_1$ and $\widehat{X}_2$ to ensure that $\|X_1\|_F = \|Z\|_F$ and $\|X_2\|_F = \|Z\|_F$. We then return the solution that minimizes $\phi(X, Z, R, \Lambda)$. In the below code, `xx1` denotes X_1, `xx2` denotes X_2, `ZR1` denotes Y, and `ZR2` denotes Y'

```
def find_soft_clusters(Z, R, a):
    ZR1 = Z.dot(R)
    ZR2, Rn = flip_negative_columns(ZR1)
    for st in range(0, 2):
        ZR = ZR1 if st == 0 else ZR2
        ZR[ZR < 0] = 0
        n = ZR.shape[0]
        X1 = ZR
        for i in range(0, n):
            X1[i, :] = X1[i, :]/np.sum(X1[i, :])
        if st == 0:
```

```
            XX1 = a * X1
        else:
            XX2 = a * X1

    n1 = norm_diff(XX1, ZR1, np.eye(ZR1.shape[1]))
    n2 = norm_diff(XX2, ZR2, np.eye(ZR2.shape[1]))
    X = XX2 if n1 > n2 else XX1
    return [X, Rn]
```

2.4.10 Computing a Flexible Clusters: "Flex Clustering"

The final alternative to the hard clustering procedure is the so-called "flex clustering" algorithm which computes a solution X^* that allows for membership to multiple clusters. As in the soft clustering case, we simply replace the procedure to minimize $\phi(X, Z, R, \Lambda)$ with fixed Z, R, Λ.

In fact, the flex clustering procedure is identical to the hard clustering feature save for the 1-shifting sub-routine which attempted to shift columns with extraneous 1's to columns with a sum of 0. Recall the definition of a solution X where X is defined for all $i \in V$, $j \in [1, k]$ as:

$$X_{ij} = \begin{cases} 1 & i \in C_j \\ 0 & otherwise \end{cases}$$

Then at a high level, the absence of the 1-shifting procedure allows for membership to multiple clusters at the risk of an "unbalanced" clustering solution (i.e. the size of clusters may vary more than in the hard clustering case). The below code represents the flexible clustering procedure to find a solution X that minimizes $\phi(X, Z, R, \Lambda)$ with fixed Z, R, Λ. Variable naming is the same as in the soft clustering case.

```
def find_flex_clusters(z, r, a):
    zr1 = z.dot(r)
    k = zr1.shape[1]
    zr2, rn = flip_negative_columns(zr1)
    for st in range(0, 2):
        zr = zr1 if st == 0 else zr2
        n = zr.shape[0]
        x1 = np.zeros((n, k))
        J = np.argmax(zr.H, axis=0)
        for i in range(0, n):
            x1[i, J[0, i]] = 1

        if st == 1:
            xx1 = a*x1
        else:
            xx2 = a*x1

    n1 = norm_diff(xx1, zr1, np.eye(zr1.shape[1]))
    n2 = norm_diff(xx2, zr2, np.eye(zr2.shape[1]))
    x = xx2 if n1 > n2 else xx1
    return [x, rn]
```

3 Recipe Rating Prediction

3.1 Motivation

In this section, we investigate the problem of predicting the rating of a recipe given only the list of ingredients used in the recipe. Given that a recipe comprises a set of ingredients, their quantities,

and step-by-step instructions for the preparation and usage of the ingredients, it is natural to assume that there exist a relationship between the set of ingredients in a recipe and the quality of a recipe.

In particular, we aim to investigate if the pairwise *relationships* between ingredients has any bearing on a recipe's rating. As an example, consider the following situation: say one is presented with two pairs of ingredients, respectively chocolate chips and marshmallows and celery and cake flour. Now, if told that the first pair belongs to recipe R_1 and the latter pair belongs to recipe R_2 and asked to predict which of R_1 or R_2 has a higher rating, it seems reasonable to assume that R_1 has a higher rating given the natural cohesiveness of chocolate chips and marshmallows especially when compared with the dissonance of celery and cake flour.

However, it remains undefined what lends two ingredients said "cohesiveness" or "dissonance." Therefore, the aim of this paper is two-fold: firstly, to determine a meaningful metric for the pair-wise harmony of ingredients and secondly, if the pair-wise harmonies of all the ingredients in a recipe influence the overall quality of the recipe.

We investigate both these aims through the application of graph clustering to recipe prediction. Let R denote a set of recipes and let $I(R)$ denote the set of unique ingredients used in R. Also, let $\sigma : (I \times I) \rightarrow \mathbb{R}^{[-1,1]}$ be a "harmony" function where $\sigma(i, j)$ denotes the relative harmony of ingredients i and j such that positive values signify mutual compatibility and negative values signify dissonance.

We can thus model $I(R)$ as a weighted undirected graph $G(I(R), W)$ where the vertices of G are the unique ingredients in $I(R)$ for every $i, j \in I(R)$:

$$W_{ij} = \sigma(i, j)$$

Then, we can partition the ingredients into clusters of harmonious ingredients where ingredients in distinct clusters are mutually incompatible and ingredients within the same cluster are synergistic.

These clusters then serve as features for various machine learning algorithms to predict recipe ratings. We hypothesize that should these clusters allow for better-than-chance predictions, then we can conclude both the validity of the chosen σ and that the pairwise-relationships between ingredients do indeed influence the rating of a recipe.

3.2 Dataset

3.2.1 Source

All recipe data was sourced from allrecipes.com, a popular recipe database site. Since allrecipes.com allows for user-submitted recipes and user ratings of recipes, its huge user-base lends itself to a large volume of recipes and a large volume of ratings. This was our main motivation in selecting allrecipes.com. A secondary motivation was the high precision of its ratings (the ratings are precise down to the 10^{-8}). We downloaded 19,645 recipes containing 7170 unique ingredients. As the histogram above demonstrates, the ratings were heavily biased towards 4's and 5's.

3.2.2 Collection

Each recipe on allrecipes.com is identified by a unique positive integer i such that the recipe is accessible at the URL http://allrecipes.com/recipe/i. The first iteration of the recipe downloader simply iterated over identifiers in the range $[1, 10^7]$ and attempted to download the recipe at each URL, ignoring the URL if it produces a URL error or socket error.

However, the downloader could only process 100 recipes before its IP address was blocked. To remedy this issue, we attempted spoofing the GoogleBot site indexing user agent, adding random delays between requests, and randomizing the order in which the identifiers were processed. However, none of these methods proved fruitful.

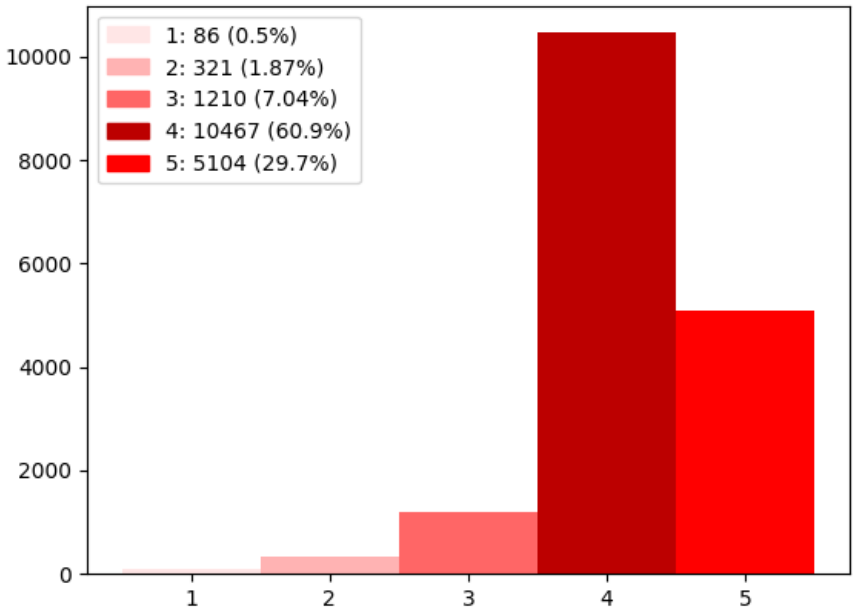

Figure 1: Histogram of Recipe Ratings

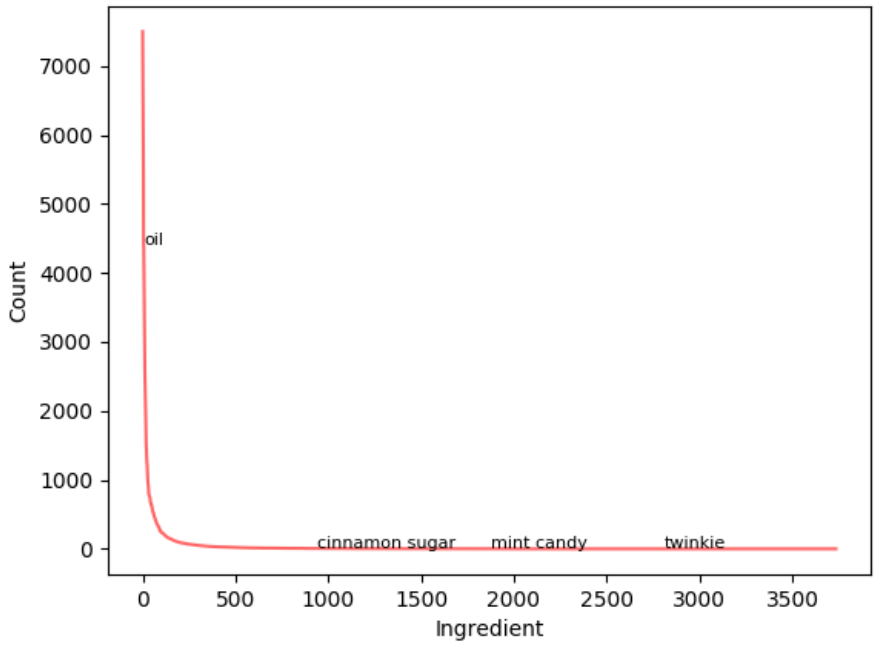

Figure 2: Distribution of Ingredient Frequencies

Original		*Tokenized & Tagged*
"semisweet chocolate chips"	→	{('semisweet', 'JJ'), ('chocolate', 'NN'), ('chips', 'NNS')}

↓

Final		*Lemmatized Nouns*
"chocolate chip"	←	{'chocolate', 'chip'}

Table 1: Pre-Processing Example: "semisweet chocolate chips" is first tokenized and part-of-speech tagged. Then, the adjective component "semisweet" is eliminated and each noun component is lemmatized, leaving "chocolate" unchanged but transforming "chips" into "chip." Finally, the lemmatized noun components are joined resulting in "chocolate chip."

Our solution was to spawn 10 AWS instances and download recipes in parallel across the instances in mini-batches of 30 recipes. To automate this process, we wrote a Python script that took as input a file containing the IP addresses of the instances, a delay t (in milliseconds), and a batch size s and proceeded to pull the latest version of the recipe parsing code to each instance, run the recipe parser in batches of s separated by t ms, and copy the resulting XML files from each instance to a local directory. The entire process took approximately 3 hours with $t = 1000$ ms and $s = 30$.

3.2.3 Preprocessing

We implemented a Python recipe parser script that took as input a directory containing the XML files created by the recipe downloader and outputted an affinity matrix W of size $m \times m$ where $m \in \mathbb{Z}^+$ denotes the number of unique ingredients contained in the XML files. For each recipe r, denote its set of ingredients as $I(r)$ and its rating as $S(r)$. To process each recipe r, for each ingredient in $I(r)$, we tokenize the ingredient and part-of-speech tag each component, remove all non-noun components, and then lemmatize each noun component. We then add each cleaned ingredient to a set I of ingredients and add $I(r)$ to a set R of recipes. We also disregard duplicate recipes.

In addition, we only consider recipes where $S(r) > 0$ as $S(r) = 0$ if and only if r has not been rated. Since we take recipes as our training data and their ratings as our labels, a recipe is useless without any ratings.

Let I^* denote the entire set of unique ingredients used in R. While processing recipes, we also maintain a map $DF : I^* \to \mathbb{N}$ where $DF[i]$ denotes the frequency of ingredient $i \in I^*$ in the entire corpus of recipes we have downloaded, R.

Our recipe parser also took as input an $\alpha \in (0, 1]$ where we kept only the top α percent of unique ingredients by frequency. This allowed us to discard noisy recipes with only a few ratings.

3.3 Affinity Matrices

To cluster the ingredients, we first must generate an affinity matrix W of size $m \times m$ where $m = |I^*|$ that encodes some meaningful information regarding the relationship between ingredients. The effectiveness of the clustering algorithm is entirely dependent on the expressiveness of the affinity matrix and so we employed a variety of different techniques to generate different affinity matrices.

Recall the ingredient graph $G(I(R), W)$ from section 3.1 where:

$$W_{ij} = \sigma(i, j)$$

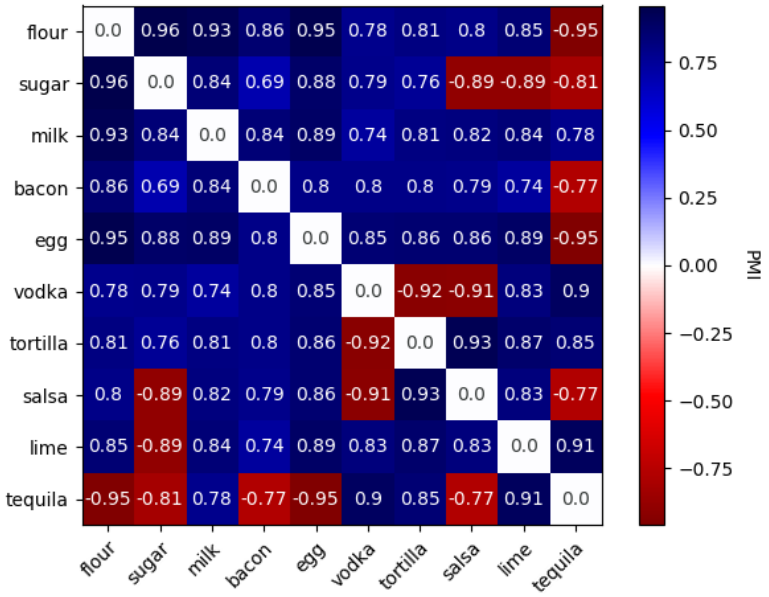

Figure 3: Example PMI Affinities

for all $i, j \in I(R)$ and $\sigma : (I \times I) \to \mathbb{R}^{[-1,1]}$ is a "harmony" function denoting the relative harmony of ingredients. Then, the affinity matrices we explore in this section exactly correspond to this W and so we are in fact exploring the space of harmony functions with the aim of finding an expressive harmony function that will result in meaningful clusters of ingredients.

3.3.1 Pointwise Mutual Information (PMI)

Our first attempt at a harmony function was to use pointwise mutual information (PMI) as a measure of co-occurrence frequency with the thought that ingredients that pair well together co-occur with greater frequency. Thus, the affinity matrix W is defined by:

$$P(i) = \frac{DF(i)}{|R|}$$

$$P(j) = \frac{DF(j)}{|R|}$$

$$P(i,j) = \frac{DF(i,j)}{|R|}$$

$$W_{ij} = PMI(i,j) = \log \frac{p(i,j)}{p(i)p(j)}$$

However, as explained in Bouma [2], PMI gives comparatively higher scores to lower frequency events. We see this in the heat-map where lime and bacon have a PMI of 0.74 whereas lime and tequila receive a score of 0.91 even though lime and tequila co-occur with magnitudes greater frequency than lime and bacon. Thus, we discarded the PMI approach.

3.3.2 Normalized Pointwise Mutual Information (NPMI)

To mitigate the low frequency bias of PMI, we turned to normalized PMI as defined in Bouma (2015). The affinity matrix, then, is given by:

$$P(i) = \frac{DF(i)}{|R|}$$

$$P(j) = \frac{DF(j)}{|R|}$$

$$P(i,j) = \frac{DF(i,j)}{|R|}$$

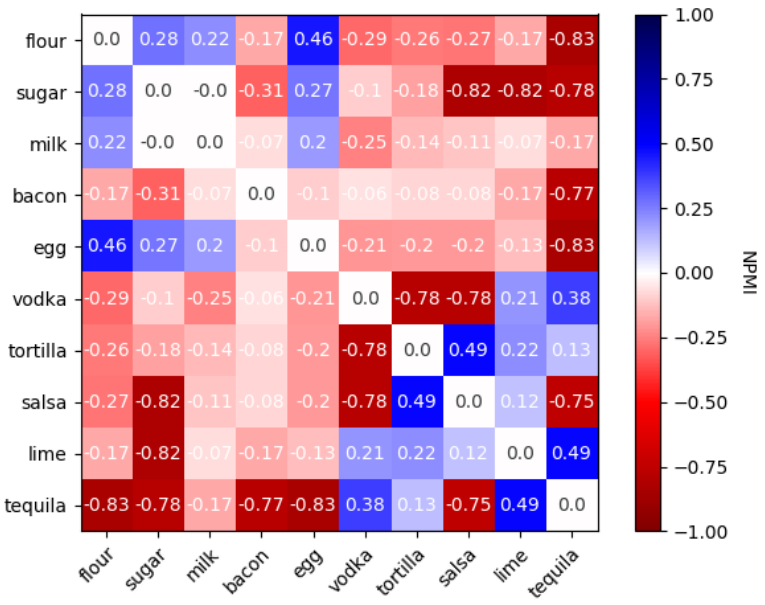

Figure 4: Example NPMI Affinities

$$PMI(i,j) = \log \frac{p(i,j)}{p(i)p(j)}$$

$$W_{ij} = \frac{PMI(i,j)}{-\log(p(i,j))}$$

As the heatmap demonstrates, the NPMI approach indeed generates meaningful harmony scores. A high-level example: lime pairs well with tortillas, salsa, and tequila, whereas flour pairs well with sugar and egg. Contrarily, tequila contrasts with flour, sugar, bacon, and egg, and vodka contrasts with tortillas and salsa.

However, NPMI only factors in co-occurrence information and thus completely discards our rating information. This is not necessarily a mark against the NPMI approach but perhaps indicates that a better harmony function exists that accounts for both co-occurrence and rating information.

3.3.3 Normalized Average Ratings

The normalized average ratings method computes pair-wise harmony scores by examining the average ratings of recipes that two ingredients co-occur in. Specifically, for two ingredients $i, j \in I$, we define W_{ij} as:

$$W_{ij} = \frac{1}{DF(i,j)} \sum_{r \in R \mid i,j \in r} S(r)$$

However, before we compute W, we first calculate μ_S, the mean rating, and replace every $s \in S$ with s' where s' is defined as:

$$s' = s - \mu_S$$

Our intention in subtracting μ_S from every rating was to remove the rating bias towards ratings of 4 and 5 and to denote mathematically that ratings of 1, 2, and 3 are considered "poor" and should thus have negative impact on the harmony score. In our particular dataset, we found μ_S to be 4.2090774702882605.

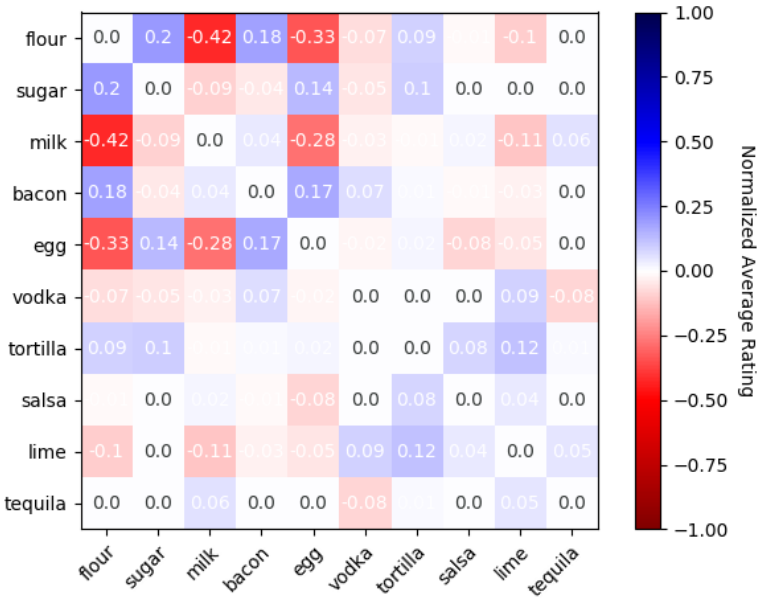

Figure 5: Example Averaged Normalized Rating Affinities

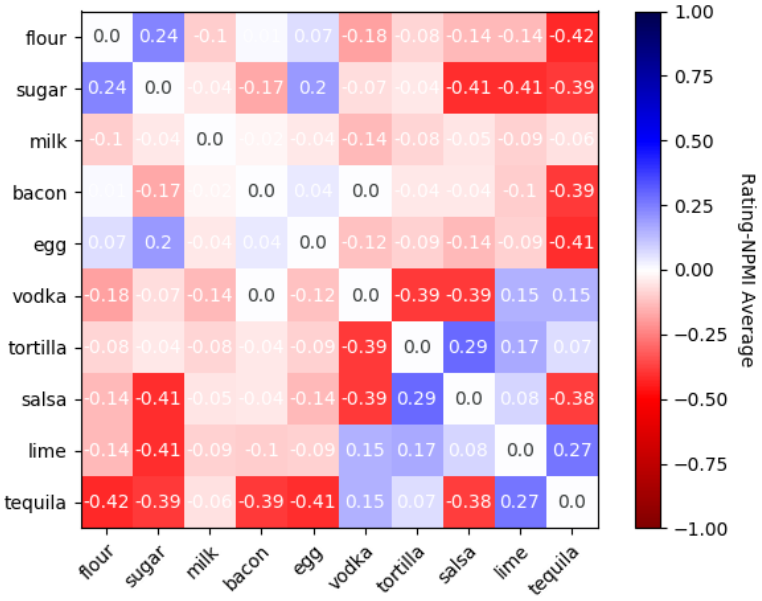

Figure 6: Example NPMI-Ratings Hybrid Affinities

3.3.4 NPMI Ratings Hybrid

To account for both co-occurrence and rating information, the NPMI-Ratings hybrid approach simply entails taking the arithmetic mean of NPMI and normalized average rating scores, defining W_{ij} as:

$$W_{ij} = \frac{1}{2}(NPMI(i,j))(\frac{1}{DF(i,j)} \sum_{r \in R \mid i,j \in r} S(r))$$

chocolate cookies	tomato	potato	chow mein noodles	lime	sake
packaged crumb crust	red onion	onion	baby bok choy	cachaca	mochi
red raspberry jam	basil	leg of lamb meat	herb blend	silver tequila	sheet nori
chocolate wafer cookie	tagliatelle	chicken or lamb stock	stir fry vegetables	diet ginger ale	dashi kombu
brown sugar substitute	linguini	carrots lengthwise	classic stir fry sauce	banana	bonito shavings
shortbread pie crust	dandelion greens		asparagus tips	sour cocktail mix	naruto
				margarita salt	yuzu

Table 2: Sample Clusters ($k = 800$, $|I| = 6690$)

SVM	
C	0.1
kernel	rbf
gamma	$'auto'$

Random Forest	
n_estimators	178
max_features	'auto'
max_depth	40

Logistic Regression	
C	0.001
penalty	L2

Table 3: Hyperparameters

3.4 Clusters

We selected the NPMI affinity matrix as it gave the most qualitatively meaningful pairwise harmony scores. Using the affinity matrix, we perform the signed spectral clustering algorithm from section 2 with the goal of producing clusters of complementary ingredients.

As the below clusters indicate, the NPMI affinity matrix produces extremely identifiable clusters. For instance, the clusters seem to respectively represent: baked desserts, pasta dishes, stews, stir fries, cocktails, and Japanese foods.

3.5 Features

Since the purpose of the rating prediction application is to validate the clustering algorithm, we selected features that directly relate to the clustering.

Let $C_1, C_2, ..., C_k$ denote the k clusters into which we partitioned I. Then let $C_i^{(r)} \subseteq C_i$ ($i \in [1, k]$) be the subset of ingredients in recipe r contained in cluster C_i. Then, we have $3k$ features $|C_i^{(r)}|$, assoc$(C_i^{(r)})$ and cut$(C_i^{(r)})$ ($i \in [1, k]$) where:

$$\text{links}(A, B) = \sum_{i \in A,\ j \in B} W_{ij}$$

$$\text{assoc}(C_i^{(r)}) = \text{links}(C_i^{(r)}, C_i^{(r)}) = \sum_{i,j \in C_i^{(r)}} W_{ij}$$

$$\text{cut}(C_i^{(r)}) = \text{links}(C_i^{(r)}, \overline{C}_i^{(r)}) = \sum_{i \in C_i^{(r)},\ j \in \overline{C}_i^{(r)}} W_{ij}$$

3.6 Classifiers

Following initial experiments to select the most promising classifiers, we chose Support Vector Machines, Random Forests, and Logistic Regression. We used the scikit-learn implementation of all three classifiers with the following parameters (determined through randomized search):

We also tested ordinal logistic regression but found its results were comparable to standard logistic regression.

3.7 Cluster Hyperparameters

We first needed to determine the number of clusters to partition I into that would yield the optimal classifier performance. To do so, we evaluated an untuned random forest classifier over a range of potential k values. We used a processed ingredient set of size $|I| = 3737$ and tested

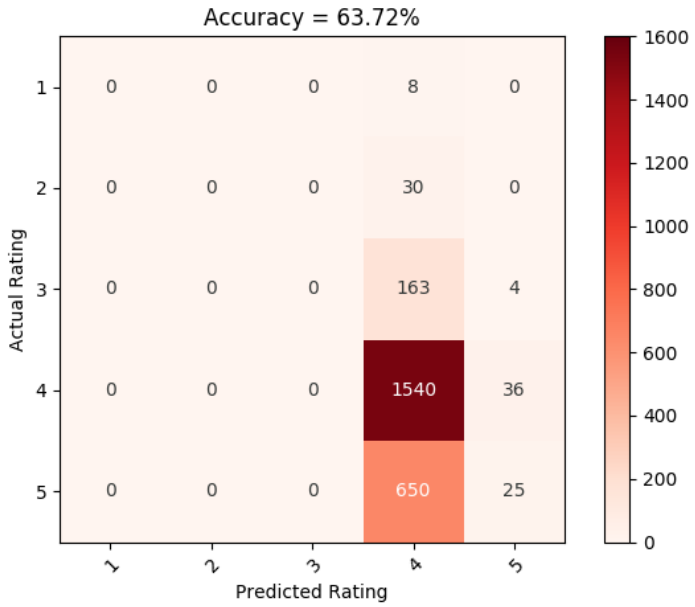

Figure 7: SVM Confusion Matrix

fourteen candidate cluster sizes evenly spaced in the range $[1, 3737]$. The specific values tested were: $1, 267, 534, 799, 1065, 1331, 1597, 1863, 2129, 2395, 2661, 2927, 3193, 3459$, and 3725. We found that $k = 534$ gave the best results.

3.8 Baseline

Simply picking the majority label (rating 4 in our case) would yield an accuracy of 60.9%.

3.9 Results

We evaluated the three models using the features that resulted from clustering the $|I| = 3737$ ingredients into $k = 534$ clusters. The random forest model clearly outperformed the logistic regression and SVM classifiers with an accuracy of 67.88%, a modest increase over baseline accuracy.

All three classifiers performed nearly perfectly on recipes with ratings of 4 which is to be expected given the skew of our dataset towards 4-star recipes. Additionally, the three classifiers all mis-predicted the 1 and 2-star recipes as 4-stars, which is also to be expected given that 1 and 2-star recipes collectively make up 2.37% of the data-set.

Classifier performance primarily varied in the classification of 4 and 5-star recipes with all three classifiers correctly predicting most 4-star recipes and mispredicting the majority of 5-star recipes. However, the random forest did a decent job at predicting 5-star recipes with an accuracy of 14%.

4 Conclusions and Future Work

In this paper, we explored various approaches for assessing the pairwise harmony of ingredients and predicting the ratings of recipes using ingredient clusters. The difficulty of this prediction problem derives from a number of factors; for one, recipe rating may not directly correlate to the actual quality of the recipe. Furthermore, there are a number of other qualities inherent to a recipe that we ignore including ingredient quantities, preparation methods, cooking time, and nutritional information among others.

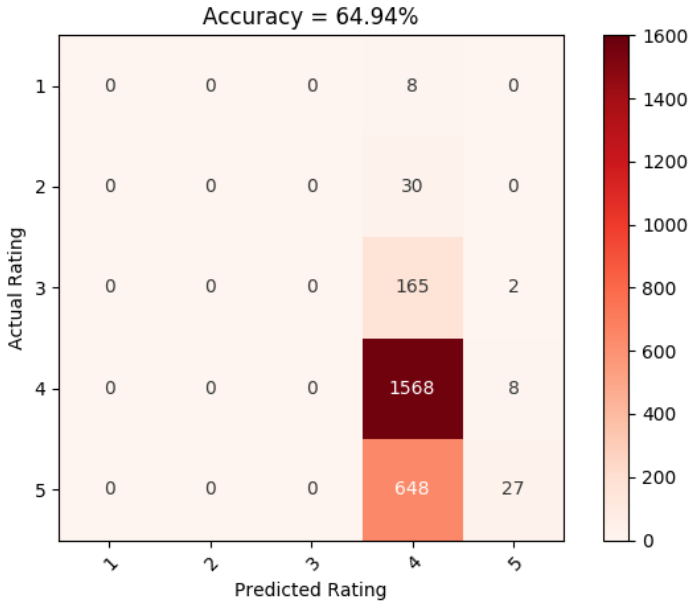

Figure 8: Logistic Regression Confusion Matrix

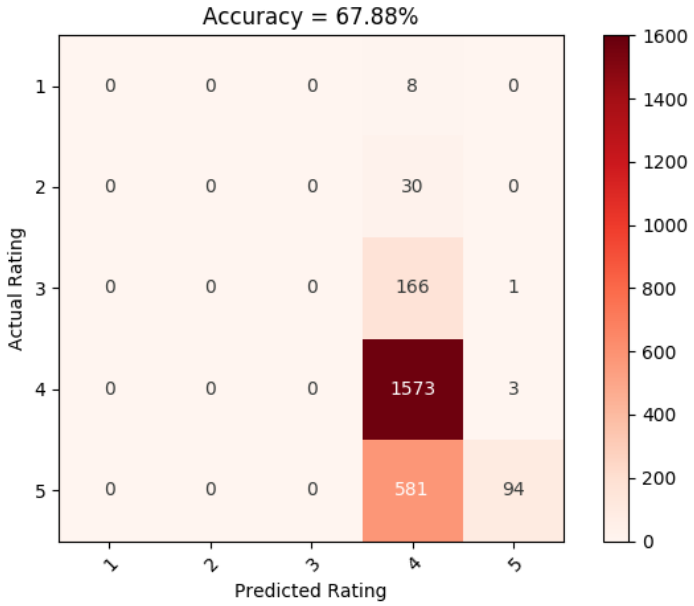

Figure 9: Random Forest Confusion Matrix

However, we did achieve a classification accuracy that exceeded baseline accuracy by a not-insignificant amount. This indicates that the relative harmony of ingredients does indeed impact the quality of a recipe. Logically, a novel recipes could be generated by simply selecting a set of ingredients that pair well together. Thus, we could investigate recipe generation procedures such as performing random walks through the graph represented by the affinity matrix.

In addition, further investigation into meaningful harmony metrics is warranted. One potential approach is using the cosine similarity between ingredient word embeddings as a harmony metric. Another approach could factor in the sentiment of the reviews themselves.

Determining a meaningful harmony metric has applications beyond recipe rating prediction in recipe generation, recipe recommendation (on a cooking website), or ingredient substitution suggestions (when one lacks an ingredient or cannot consume an ingredient for some reason).

References

[1] Yong-Yeol Ahn et al. "Flavor network and the principles of food pairing". In: 1 (Nov. 2011).

[2] Gerlof Bouma. "Normalized (Pointwise) Mutual Information in Collocation Extraction". In: (Jan. 2009).

[3] Jill Freyne and Shlomo Berkovsky. "Intelligent Food Planning: Personalized Recipe Recommendation". In: *Proceedings of the 15th International Conference on Intelligent User Interfaces*. IUI '10. Hong Kong, China: ACM, 2010, pp. 321–324. ISBN: 978-1-60558-515-4. DOI: 10.1145/1719970.1720021. URL: http://doi.acm.org/10.1145/1719970.1720021.

[4] Jean Gallier. "Spectral Theory of Unsigned and Signed Graphs. Applications to Graph Clustering: a Survey". In: *CoRR* abs/1601.04692 (2016). arXiv: 1601.04692. URL: http://arxiv.org/abs/1601.04692.

[5] João Sedoc and Aline Normoyle. "Seating Assignment Using Constrained Signed Spectral Clustering". In: *CoRR* abs/1708.00898 (2017). arXiv: 1708.00898. URL: http://arxiv.org/abs/1708.00898.

[6] João Sedoc et al. "Semantic Word Clusters Using Signed Normalized Graph Cuts". In: *CoRR* abs/1601.05403 (2016). arXiv: 1601.05403. URL: http://arxiv.org/abs/1601.05403.