UNIVERSIDAD DE GRANADA

Práctica 1.b

Técnicas de búsqueda local y algoritmos greedy para el problema del aprendizaje de pesos en características (APC).

Mario Román García

28 de marzo de 2018

- DNI: 77145669N
- Email: mromang08@correo.ugr.es

- Grupo 1 (Lunes de 17:30 a 19:30)
- Algoritmos: 1-NN, RELIEF, Búsqueda Local.
- Metaheurísticas, 2017-2018.

Índice

1. Formulación del problema

Trataremos un problema de **aprendizaje de pesos en características** (APC), consistente en la optimización de la simplicidad y precisión de un clasificador 1-NN; es decir, un clasificador que asigna a cada instancia la clase de la instancia más cercana a él, para una distancia euclídea modificada por un vector de pesos. Así, cada solución del problema vendrá dada por un vector de valores reales $w_i \in [0, 1]$ para $1 \leq i \leq n$, donde n es el número de características que tiene el problema. La bondad de un clasificador de este tipo sobre un conjunto de evaluación T vendrá determinada como

$$F(\{w_i\}) = \alpha \text{tasaClas}\,\{w_i\} + (1 - \alpha)\text{tasaRed}\,\{w_i\}\,.$$

En esta fórmula, *tasaClas* debe ser entendida como la **precisión** del algoritmo, midiendo el porcentaje de aciertos del clasificador en el conjunto de evaluación T. Por otro lado, *tasaRed* debe ser entendido como la **simplicidad** de la solución, que mide el número de pesos w_i que quedan por debajo de 0.2, y que consecuentemente no se tienen en cuenta al calcular las distancias. En nuestro caso tenemos determinado un valor de $\alpha = 0.5$ que equipara ambas métricas.

En resumen, el algoritmo deberá tomar un conjunto de datos y producir un clasificador 1-NN con unos pesos asignados a las características que sean sencillos, en el sentido de usar el mínimo número de características, y que sean precisos, en cuanto a su habilidad para clasificar instancias fuera del conjunto de entrenamiento.

2. Aplicación de algoritmos

2.1. Esquema de representación de soluciones

Como hemos tratado anteriormente, una solución del problema viene dada por un vector de pesos. Nuestra representación específica de una solución vendrá dada por una lista de valores reales.

$$\{w_i\}_{0 \leq i \leq n} = (w_1, w_2, \ldots, w_n)$$

Donde n será el número de características que tenga nuestro conjunto de datos y w_i. En nuestra implementación específica, los pesos vendrán dados por un vector contiguo en memoria de valores en coma flotante de precisión doble en 64 bits; para facilitar su tratamiento, en ciertos algoritmos se traducirán a listas enlazadas, pero su interpretación matemática será siempre constante y se describe explícitamente en la función objetivo.

2.2. Clasificador 1-NN, función objetivo

Se define la **distancia con pesos** $\{w_i\}$ entre dos vectores t y s como

$$\text{dist}(w, t, s) = \sum_{i=0}^{n} w_i (t_i - s_i)^2.$$

Y nuestro clasificador kNN para unos pesos $\{w_i\}$ consiste en devolver la clase del punto que minimiza la distancia. Es decir, es una implementación de un clasificador 1NN, que para cada instancia devuelve la clase de su vecino más cercano. En el caso de la función objetivo, al no tener un conjunto de test separado, usamos la técnica de *leave-one-out*.

Algorithm 1 Función objetivo (w : Pesos, T : Training)

1: $\text{Obj}(w, T) = \alpha \cdot \text{precision}(w, T) + (1 - \alpha)\text{simplicity}(w)$
2: $\text{TasaRed}(w) = \text{length}[x < 0.2 \mid x \in w]/\text{length } w$
3: $\text{TasaClas}(w, T) = \sum_{t \in T}(\text{knn}(w, T - t, t) == s.Clase)/\text{length } s$
4: $\text{knn}(w, T, t) = (\text{minimizador}_{t' \in T}(\text{dist}^2(\text{trunca}(w), t', t))).Clase$
5: $\text{trunca}(w) = \{0 \text{ si } w_i < 0.2; \quad w_i \text{ en otro caso } \mid w_i \in w\}$

En el código original esta función objetivo aparecerá implementada dos veces: una vez para el puntuador de los algoritmos y otra vez para la función objetivo. Esta duplicación tiene como ventaja que separa completamente las partes de evaluación del código de los algoritmos, reduciendo la posibilidad de error. Además,

- la implementación para evaluación es corta y es más fácil de verificar que está escrita correctamente;
- mientras que la implementación de la función objetivo está fuertemente optimizada, usando paralelismo, pero en caso de que tuviera cualquier error, eso no se vería reflejado en las puntuaciones de los algoritmos.

Es importante notar que la formulación original del problema no asume que todas las variables sean reales, y en el caso en el que son discretas asume una distancia de Hamming que simplemente indica si las dos características son exactamente iguales.

$$\text{dist}(a, b) = \delta_{ab} = \begin{cases} 1 & \text{si } a = b \\ 0 & \text{si } a \neq b \end{cases}$$

En los conjuntos de datos que trataremos en este análisis, tenemos de hecho que todas las características vienen dadas por reales, y nuestra implementación, aunque sería fácilmente extensible, no tratará el otro caso explícitamente.

3. Pseudocódigo de los algoritmos de búsqueda

Al usar pseudocódigo nótese que para reflejar más fielmente nuestra implementación, que intenta ser declarativa y basada en el paradigma de **programación funcional**, usaremos pseudocódigo basado en definiciones declarativas de funciones matemáticas.

3.1. 1NN

La primera solución, que usaremos como referencia, es completamente trivial y se basa simplemente en usar directamente el clasificador 1NN con una distancia euclídea

usual. Dentro de nuestra formulación del problema, esto equivale a una solución que simplemente devuelva en todos los casos un vector de pesos hecho constantemente de unos.

Algorithm 2 1NN (t : Training)

1: $\text{1NN}(t) = \text{replica } (\text{nAttr}(t)) \text{ veces } 1$

3.2. Relief

La segunda solución de referencia implementa una variante del algoritmo greedy RELIEF [KR92]. En esencia, para cada instancia calcularemos la distancia al amigo (instancia con la misma clase) más cercano y al enemigo (instancia con distinta clase) más cercano y actualizaremos el vector de pesos en consecuencia.

Tenemos una función "normaliza", que se aplicará al resultado final, dividirá todos los pesos por el máximo y asignará cero a aquellos que fueran negativos.

Algorithm 3 RELIEF (s : Semilla, T : Training)

1: $\text{Relief}(s, T) = \text{normaliza } \left(\sum_{t \in T} \text{vectorDistE}(t) - \text{vectorDistA}(t)\right)$
2: $\text{normaliza}(w) = \left\{w_i^+ / \text{máx}(w) \mid w_i \in w\right\}$
3: $\text{vectorDistA}(t) = \text{map valorAbsoluto } (t - \text{amigoMasCer}(t))$
4: $\text{vectorDistE}(t) = \text{map valorAbsoluto } (t - \text{enemigoMasCer}(t))$
5: $\text{amigoMasCer}(t) = \text{minimizador}_{t'.Clase=t.Clase} \left(\text{dist}^2(t', t)\right)$
6: $\text{enemigoMasCer}(t) = \text{minimizador}_{t'.Clase \neq t.Clase} \left(\text{dist}^2(t', t)\right)$

3.3. Búsqueda local

El operador de **generación de vecinos** será una variación que tomará aleatoriamente un índice extraído de una distribución uniforme y un epsilon extraído de una distribución normal.

Algorithm 4 Vecinos en búsqueda local (w : Pesos, i : Índice, ε : Epsilon)

1: $\text{Vecino}(\varepsilon, i, w) = \text{truncaEntre0y1 map } (\lambda(x, i).x + \delta_{ij}\varepsilon) \text{ (indexa } w)$

El método de búsqueda consiste principalmente en dos funciones. Una de ellas busca una mejora local, aplicando repetidamente la generación de vecinos con argumentos procedentes de una distribución normal y una permutación aleatoria y la otra ejecuta varias veces la búsqueda local. Lo usaremos como **exploración del entorno**. Nótese que esta exploración del entorno sigue la técnica del *primero mejor* en lugar de generar un número fijo de variaciones locales y elegir la mejor entre todas ellas.

Para esta exploración tendremos una estructura de datos dada por

- los pasos en total dados hasta el momento,

- los pasos dados desde la última optimización,
- un generador aleatorio,
- una permutación aleatoria de los índices que se generará según sea necesaria para tomar índices aleatorios,
- la mejor solución hasta el momento, y
- la bondad de esa solución según la función objetivo.

Habrá una función de exploración que actualice esta estructura y otra función que controlará el número de veces que exploramos el entorno.

Algorithm 5 Búsqueda Local (s : Semilla, t : Training)

1: busqueda(s, t) = hastaQue(pasos$_{\text{glob}} = 15000$ o pasos$_{\text{loc}} = 20 \cdot n$) aplica explora a solInicial
2: explora$(pasos, w)$ = minimizadorDe $(\lambda w.\text{objetivo}(w))$ entre $\{w, \text{Vecino}(\varepsilon, i, w)\}$
3: $\varepsilon_1, \varepsilon_2, \cdots =$ random$\mathcal{N}(\mu = 0.5, \sigma = 0.5)$
4: $i_1, i_2, \cdots =$ randomPermutacion

La generación de la **solución aleatoria inicial** se hace directamente usando las librerías del lenguaje [Buc11], que proporcionan funciones para crear listas potencialmente infinitas de reales distribuidos respecto a una distribución normal dada. Internamente, se usa el método de Box-Müller [BM+58] para generar los valores. De esa lista extraemos sólo los números necesarios para construir una solución.

Algorithm 6 Solución inicial (t : Training)

1: solInicial(t) = tomaLos (nAttr(t)) primerosDe random$\mathcal{N}(\mu = 0.5, \sigma = 0.5)$

3.3.1. Variante de la búsqueda local

Como añadido a la práctica, implementamos una variante de la búsqueda local que en lugar de variar una sola dimensión a cada paso, varía el vector completo. Lo único que cambia respecto a la búsqueda local anterior es por tanto el operador de variación, que ahora necesita un vector aleatorio generado de acuerdo a $\mathcal{N}(0, \sigma)$. Además, pasaremos a truncar entre 0.2 y 1, para evitar que regresen al vector características que queríamos eliminar, y pasamos a usar una varianza ligeramente menor, que elegimos empíricamente en $\sigma = 0.2$.

Algorithm 7 Vecino modificado (w : Pesos, i : Índice, $v\varepsilon$: vectorAleatorio)

1: Vecino(ε, i, w) = truncaEntre0.2y1 $(w + v\varepsilon)$ para $v\varepsilon \sim \mathcal{N}(0, 0.2)$

4. Pseudocódigo de los algoritmos de comparación

Para comparar los algoritmos entre sí usaremos validación cruzada en 5 partes. Tendremos un programa que parte los conjuntos de datos en cinco subconjuntos balanceados. Nótese que la proporción entre clases se mantiene igual a la del conjunto original al partir los datos inicialmente en dos bloques según su clase, partir en cinco partes cada uno de los trozos y luego recomponer las partes globalmente.

Algorithm 8 Partición en 5 (t : Training)

1: 5split(t) = une partes1 partes2
2: deClase1 = filtra (.Clase $\equiv$ 1)
3: deClase2 = filtra (.Clase $\equiv$ 2)
4: partes1 = parteEnTrozosDe $\lceil$longitud(t)/5$\rceil$ deClase1
5: partes2 = parteEnTrozosDe $\lceil$longitud(t)/5$\rceil$ deClase2

La regla que indica cómo debemos usar estas particiones simplemente indica que cada una de las partes sirve para validar las otras cuatro; y que por tanto, para cada una de ellas, debemos entrenar el clasificador con las otras cuatro y luego aplicarlas.

Algorithm 9 Comparación (algoritmo)

1: **Para cada** parte $p \in \{p1, p2, p3, p4, p5\}$
2: *pesos* $\leftarrow$ Ejecuta *algoritmo* en $\{p1, p2, p3, p4, p5\} - p$
3: Puntúa 1-NN con *pesos* y entrenamiento ($\{p1, p2, p3, p4, p5\} - p$) sobre p

La implementación de este algoritmo de comparación es parte del archivo `make` que permite reproducir la práctica que se comenta en la próxima sección.

Finalmente, la puntuación se hace aplicando 1-NN directamente, de forma similar a como lo hacíamos para calcular la función objetivo.

Algorithm 10 Puntuación (w : Pesos, T : Training, S : Test)

1: Obj$(w, T, S) = \alpha \cdot$ precision$(w, T, S) + (1 - \alpha)$simplicity(w)
2: TasaRed(w) = length$[x < 0.2 \mid x \in w]$/length w
3: TasaClas$(w, T, S) = \sum_{s \in S}$(knn$(w, T, s) == s.Clase$)/length s
4: knn$(w, T, s) =$ (minimizador$_{t \in T}$(dist^2(trunca$(w), t, s)$)).$Clase$
5: trunca$(w) = \{0$ si $w_i < 0.2$; $\quad w_i$ en otro caso $\mid w_i \in w\}$

5. Procedimiento considerado, manual de usuario

El código de esta práctica está escrito en el lenguaje de programación **Haskell** [P+03]. Esto nos ha permitido usar estructuras de alto nivel para las soluciones que permiten una optimización muy agresiva y que proporcionan paralelismo automáticamente [Les08]. El requisito fundamental para compilarlo es tener instalada **stack**, la herramienta de

compilación de Haskell; además de ella, usa **GNU make** [Smi92] para hacer el proceso de validación y generación de soluciones completamente reproducible.

El archivo `makefile` es el encargado de ejecutar los programas de manera acorde para conseguir los datos finales. En él se encuentran varias semillas de aleatoriedad general (`$SEEDn`) que son las que se envían a los distintos algoritmos. Este mismo archivo contiene ejemplos de llamada a los ejecutables de la práctica y permite crear las soluciones de forma reproducible. Además de todo lo que se documenta en el makefile, hemos realizado un proceso previo de limpieza de los conjuntos de datos en el que hemos eliminado todas las líneas duplicadas.

Tenemos varios ejecutables completamente indepedientes y que pueden usarse con cualquier instancia del problema que esté en formato `.arff` o con cualquier solución dada por una cabecera `@time ...` midiendo los segundos que ha tardado y una lista de valores para los pesos en formato CSV:

- `bin/fivefold`, que toma como entrada un archivo `.arff` y crea 5 archivos entre los que reparte sus instancias, de forma que queden balanceadas;
- `bin/scorer`, evalúa usando la función objetivo descrita anteriormente, recibirá el conjunto de training por la entrada estándar y tendrá como argumentos de línea de comandos al conjunto de test y la solución;
- `bin/Onenn`, implementación trivial de la solución que devuelve todos los pesos a 1;
- `bin/Relief`, implementación del algoritmo greedy Relief; y
- `bin/LocalSearch`, implementación de la búsqueda local.
- `bin/LocalSearch2`, variación de la búsqueda local.

Todas las implementaciones reciben como argumento de línea de comandos una semilla aleatoria y leen por la entrada estándar un conjunto de entrenamiento; acabarán devolviendo una solución por salida estándar.

El uso común de los programas será a traves del comando `make`. Normalmente, querremos generar un reporte de la bondad de un algoritmo determinado usando validación cruzada en cinco partes. Por ejemplo, supongamos que queremos generar un reporte de la bondad del algoritmo de búsqueda local sobre el conjunto de datos `parkinsons.arff`. Para ello lanzaremos los siguientes comandos.

```
make data/parkinsons.arff.LocalSearch.report
cat data/parkinsons.arff.LocalSearch.report
```

La ventaja de este enfoque es que permite la reutilización de los resultados ya calculados (que normalmente serán costosos en tiempo) automáticamente, así como el cálculo programado de las dependencias y cálculos estrictamente necesarios, teniendo en cuenta los ya realizados, para producir cualquier resultado concreto.

6. Experimentos y análisis de resultados

El único parámetro que nuestros algoritmos usarán globalmente es la semilla de generación aleatoria. En los experimentos que describimos aquí usaremos siempre las semillas $s = 0, 1, 2, 3, 4$ en cada una de las partes de la validación cruzada, respectivamente. Nótese que estos valores, como se ha comentado anteriormente, son argumentos a los ejecutables.

Además de ella, existen parámetros que vienen fijados por los requisitos de la práctica: la desviación típica usada en la generación de vecinos de la búsqueda local se fija siempre en $\sigma = 0.3$, y la distribución de importancia entre precisión y simplicidad se fija siempre en $\alpha = 0.5$.

6.1. 1-NN

Analizando el 1-NN, encontramos que los resultados son razonables en cuanto a precisión, pero que, al haberse conseguido a costa de usar toda la información disponible, tenemos un agregado por debajo incluso del que hubiéramos conseguido dejando todos los pesos a cero. Esta será simplemente una marca inicial sobre la que mejorar, en lugar de un primer intento.

Cuadro 1: Algoritmo 1-NN en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.766	0.000	0.383	0.032	0.850	0.000	0.425	0.006	0.815	0.000	0.407	0.016
Partición 2	0.828	0.000	0.414	0.026	0.800	0.000	0.400	0.012	0.722	0.000	0.361	0.015
Partición 3	0.797	0.000	0.398	0.048	0.825	0.000	0.412	0.010	0.815	0.000	0.407	0.013
Partición 4	0.844	0.000	0.422	0.025	0.725	0.000	0.362	0.006	0.833	0.000	0.417	0.020
Partición 5	0.844	0.000	0.422	0.028	0.714	0.000	0.357	0.007	0.686	0.000	0.343	0.017
Media	0.816	0.000	0.408	0.032	0.783	0.000	0.391	0.008	0.774	0.000	0.387	0.016

6.2. Relief

El primer algoritmo greedy proporcionaría simplicidad solo en los conjuntos en los que se dé la casualidad de que los pesos quedan por debajo del umbral que hemos marcado. En general, el rendimiento sigue siendo especialmente malo (en particular, peor que una solución con todo ceros); y lo único que hemos mejorado, no usar completamente todos los pesos, ni siquiera se refleja especialmente en la puntuación final.

Cuadro 2: Algoritmo Relief en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.750	0.014	0.382	0.114	0.900	0.000	0.450	0.028	0.796	0.000	0.398	0.049
Partición 2	0.844	0.014	0.429	0.123	0.800	0.000	0.400	0.024	0.741	0.000	0.370	0.052
Partición 3	0.797	0.014	0.405	0.120	0.825	0.000	0.412	0.022	0.815	0.000	0.407	0.052
Partición 4	0.891	0.014	0.452	0.116	0.700	0.000	0.350	0.022	0.815	0.000	0.407	0.069
Partición 5	0.812	0.014	0.413	0.117	0.743	0.000	0.371	0.024	0.667	0.000	0.333	0.055
Media	0.819	0.014	0.416	0.118	0.794	0.000	0.397	0.024	0.767	0.000	0.383	0.055

6.3. Búsqueda local

En el algoritmo de búsqueda local es donde podemos encontrar mejoras notables por primera vez. En particular, es el primero que aprovecha realmente la función objetivo para proporcionar soluciones mucho más simples que las anteriores, a costa de solo una ligera pérdida de precisión. La otra característica del algoritmo es que es el primero que realmente necesita tiempo de ejecución. Incluso tras haber usado estructuras que aprovechaban el paralelismo, tenemos tiempos por encima de los diez segundos para conjuntos de datos y criterios de parada dados. La búsqueda local, por tanto, ha servido principalmente para simplificar nuestros clasificadores; esto probablemente haya sido condicionado por el alto valor ($\alpha = 0.5$) que le otorgamos a la simplificación.

Cuadro 3: Algoritmo de Búsqueda local en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.641	0.986	0.813	12.503	0.600	0.955	0.777	1.080	0.667	0.909	0.788	11.244
Partición 2	0.641	0.972	0.806	15.306	0.250	1.000	0.625	1.964	0.481	0.977	0.729	6.867
Partición 3	0.562	0.986	0.774	19.600	0.250	1.000	0.625	1.004	0.630	0.977	0.803	11.771
Partición 4	0.609	0.917	0.763	10.658	0.625	0.909	0.767	0.999	0.704	0.977	0.840	9.518
Partición 5	0.688	0.986	0.837	18.705	0.229	1.000	0.614	0.911	0.627	0.932	0.780	7.889
Media	0.628	0.969	0.799	15.354	0.391	0.973	0.682	1.192	0.622	0.954	0.788	9.458

Al ser una búsqueda que sigue al primero mejor en lugar de al mejor de todo el vecindario, esperamos que haya servido para reducir la rapidez con la que el algoritmo converge a un mínimo local. Será interesante comparar en el futuro con algoritmos poblacionales o incluso intentar búsquedas locales multiarranque. En estos momentos sólo podemos comparar con los algoritmos de referencia y tenemos una mejora sustancial incluso mientras se pierde precisión.

6.3.1. Variante de la búsqueda local

Como añadido a la práctica, habíamos implementado una variante de la búsqueda local. Sus resultados son ligeramente mejores que los de la variante principal y al parecer, converge más rápido a una solución, gastando así menos tiempo. No se comporta mejor sin embargo en datos de mayor dimensionalidad; lo que nos indica que deberíamos tener en cuenta en la elección de este nuevo $\sigma = 0.2$ la dimensionalidad del conjunto que estamos considerando. El querer que elección aleatoria de vectores fuera sobre una esfera n-dimensional nos indica que una propuesta futura podría tomarla como $\sigma/\sqrt{n}$ para algún valor de σ.

6.4. Resultados globales

Encontramos por tanto una mejora significativa al usar métodos de búsqueda local frente a soluciones triviales como la proporcionada por el 1NN; y frente a soluciones basadas en algoritmos voraces, que ofrecen comparativamente una solución muy pobre especialmente en cuanto a simplicidad.

Cuadro 4: Algoritmo de Búsqueda local en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.719	0.722	0.720	6.891	0.925	0.955	0.940	0.853	0.759	0.841	0.800	4.206
Partición 2	0.844	0.792	0.818	9.386	0.775	0.727	0.751	0.708	0.741	0.841	0.791	7.469
Partición 3	0.656	0.917	0.786	26.540	0.250	1.000	0.625	1.161	0.741	0.864	0.802	6.259
Partición 4	0.672	0.778	0.725	11.279	0.725	0.955	0.840	0.689	0.796	0.841	0.819	7.203
Partición 5	0.797	0.792	0.794	10.963	0.600	0.909	0.755	0.931	0.804	0.886	0.845	11.213
Media	0.738	0.800	0.769	13.012	0.655	0.909	0.782	0.868	0.768	0.855	0.811	7.270

Cuadro 5: Resultados globales en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
1NN	0.816	0.000	0.408	0.032	0.783	0.000	0.391	0.008	0.774	0.000	0.387	0.016
RELIEF	0.819	0.014	0.416	0.118	0.794	0.000	0.397	0.024	0.767	0.000	0.383	0.055
BL	0.628	0.969	0.799	15.354	0.391	0.973	0.682	1.192	0.622	0.954	0.788	9.458
BL2	0.738	0.800	0.769	13.177	0.655	0.909	0.782	0.898	0.768	0.855	0.811	7.488

Referencias

[BM+58] George EP Box, Mervin E Muller, et al. A note on the generation of random normal deviates. *The annals of mathematical statistics*, 29(2):610–611, 1958.

[Buc11] Bjorn Buckwalter. Data.random.normal. *Hackage*, 2011. https://hackage.haskell.org/package/normaldistribution-1.1.0.3/docs/Data-Random-Normal.html.

[KR92] Kenji Kira and Larry A Rendell. The feature selection problem: Traditional methods and a new algorithm. In *Aaai*, volume 2, pages 129–134, 1992.

[Les08] Roman Leshchinsky. Data.vector. *Hackage*, 2008. https://hackage.haskell.org/package/vector-0.12.0.1/docs/Data-Vector.html.

[P+03] Simon Peyton-Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[Smi92] Paul Smith. Gnu make. *GNU Operating system*, 1992. https://www.gnu.org/software/make/.