Clasificación de dígitos manuscritos, una tarea de machine learning

Práctica de APA — entrega final

Roberto Fernández, Xisco Bosch, David Martín

Facultat d'Informàtica de Barcelona
Fecha de entrega: 21/01/15
Q1 2014-2015

Índice de contenido

1. Introducción

Nuestro trabajo ha consistido en desarrollar un modelo que sea capaz de clasificar dígitos manuscritos mediante *machine learning*. Los objetivos que nos hemos planteado para este proyecto consisten en:

1. Obtener un clasificador efectivo, con un error de predicción razonablemente bajo
2. Mejorar, asimilar y poner en práctica algunas técnicas y conceptos de aprendizaje automático vistas en clase
3. Ampliar y profundizar nuestro conocimiento sobre el campo de *Optical Character Recognition* (OCR)

Utilizamos el conjunto de datos "Semeion Handwritten Digit", donado al *Semeion Research Center of Sciences of Communication* (Roma, Italia) por una empresa del sector de la óptica y visión por computador. Este dataset consiste en 1593 dígitos escritos por aproximadamente 80 personas. Cada una de ellas escribió los dígitos del 0 al 9 dos veces, la primera intentando que el carácter sea lo más legible posible, y la segunda vez de forma rápida; naturalmente, éstas últimas serán las que supongan el principal reto para nuestro clasificador.

Hemos realizado unos pasos de exploración del *dataset*, que nos ayudan a comprender el tipo de problema al que nos enfrentamos, y nos proporciona ideas para extraer variables útiles con las que podamos experimentar diferentes modelos de clasificación, tanto lineales como no lineales. Finalmente, hemos decidido cuál es el mejor clasificador entre nuestra selección de modelos, y tal como detallamos en los siguientes apartados, los Support Vector Machines nos permiten obtener un modelo que se ajusta bastante bien a nuestro objetivos.

2. Contexto

El conjunto de datos "Semeion" se utilizó para medir la precisión de clasificación de dígitos en una estructura llamada MetaNet: ésta consiste en una serie de redes neuronales heterogéneas que actúan como jueces sobre una parte concreta de la entrada, y sus respuestas son consideradas según algunos parámetros de credibilidad, que dependen del historial de comportamiento de cada red (por ejemplo, la *confusion matrix* sobre el conjunto de *validation* y/o de *testing*). Estos parámetros o pesos se pueden obtener con la ecuación *Softmax* normalizada (usando las *confusion matrix*), con un algoritmo de *backpropagation* o con un método *self-reflexive*; cada opción presenta diferentes ventajas e inconvenientes que se describen más detalladamente en el documento de Buscema *et al*. De esta manera, la

salida y respuesta final al problema será una ponderación de las respuestas de los diferentes jueces.

Concretamente, para el problema de clasificación de dígitos manuscritos se utilizó MetaNet con cuatro tipos distintos de redes neuronales: *Logicon Projection Network* (una capa oculta con 40 unidades), *Back Propagation with Momentum and SoftMax* (dos capas ocultas con 40 unidades cada una), y *Learning Vector Quantization* (32 prototipos por clase de dígito). La entrada consistió en 797 caracteres destinados al aprendizaje, y los demás para *testing*, es decir, una mitad del total para cada proceso.
Entonces, los resultados de MetaNet se compararon con los de distintos tipos de redes neuronales (individuales). Vemos que, normalmente, MetaNet obtiene como mínimo el mejor resultado de las redes individuales, y en total acertó el 93.09% de los casos del conjunto de test.

3. Exploración del conjunto de datos

3.1. Preprocesado

El primer paso para obtener un buen clasificador es tratar los datos con los que trabajaremos para poder sacarle el máximo fruto posible. Lo primero que deberíamos hacer es el preproceso de datos, tratar el "*raw data*" para dejarlo en un estado en el que podamos entenderlo y utilizarlo. Sin embargo, en este problema, el *dataset* que recibimos ya está tratado previamente de la manera que comentamos a continuación.

Primero, se obtuvieron los 1593 dígitos del experimento y se escanearon con una resolución de 256 en escala de grises. Luego, cada píxel de la imagen original se escaló entre 0 y 1, dejando a 0 todos los píxeles con un valor menor o igual a 127 en la escala de grises y a 1 todos los píxeles con un valor superior a 127. Por último, la imagen se volvió a escalar a un cuadrado de 16x16, que son los 256 atributos binarios que componen nuestro *dataset*.

De esto obtenemos unos datos que están centrados respecto al eje horizontal y que están alineados a la izquierda de la imagen.

3.2. Extracción de variables

A partir de aquí, podemos centrarnos únicamente en la obtención de *features* que nos ayuden en la clasificación de los dígitos. Es decir, además de utilizar los 256 atributos binarios originales, obtendremos un seguido de atributos que representarán ciertas características de los dígitos. Hemos decidido utilizar también la matriz de 256 elementos porque de esta manera hemos conseguido mejorar los primeros resultados obtenidos en la entrega anterior; esto puede ser debido al paso de escalado y centrado de los dígitos, que permite establecer una relación entre algunos píxeles y el valor real del dígito.

A continuación explicaremos qué *features* utilizaremos:

- Número de píxeles negros en la ventana: nos indica cuántos píxeles están encendidos. Esto ayuda a identificar la complejidad del dígito y la longitud de los trazos. Por ejemplo, un 8 debería tener más píxeles negros que un 1.

- Densidad de píxeles negros entre la parte superior e inferior: nos indica si el dígito tiene más peso en la parte superior o inferior de la imagen. Por ejemplo, el 9 tiene más peso en la parte superior mientras que el 6 tiene más peso en la parte inferior. Igualmente el 3 debería tener una densidad muy cercana a 0.

- Densidad de píxeles negros entre la parte derecha e izquierda: nos indica si el dígito tiene más peso en la parte derecha o izquierda de la imagen. Por ejemplo, el 5 tendría más peso en la parte izquierda y el 3 en la parte derecha.

- Número de transiciones en dirección vertical: nos indica los cambios entre 1 y 0 en una línea vertical de la imagen. Para esto miraremos el número de transiciones de cada columna y seleccionaremos el máximo. Por ejemplo, el 9 tendría 4 transiciones.

- Sumas marginales: indica la cantidad de píxeles negros en cada columna y fila de la matriz del dígito, en total 16+16=32 variables. Por ejemplo, un 1 tendrá unas pocas columnas con valores altos, mientras que la suma marginal de las columnas de un 5 estará más repartida (con valores algo más altos en las primeras y últimas columnas).

- Número de bucles/ciclos: aproxima la cantidad de bucles del dígito. Utilizamos un algoritmo basado en BFS con una estructura Union-Find para obtener el número de componentes conexas del grafo que representa la matriz binaria, ya que los ciclos del dígito delimitan “agujeros”, tomando que dos píxeles/vértices pertenecen a la misma CC si tienen el mismo valor en la matriz. Es decir, obtenemos el número de “espacios en blanco” de un dígito, contando siempre con el marco exterior. Por ejemplo, el 1 tiene uno (marco exterior), el 6 tiene dos, y el 8 tiene tres.

- Número de puntos finales/bordes: nos indica cuántos “picos” tiene el dígito. Por ejemplo, el número 1 tendría 4 picos. Buscamos en la matriz booleana una serie de patrones para hacer *matching*, que representan los bordes o picos del dígito. Utilizamos una estructura Union-Find para agrupar los bordes más cercanos (distancia euclidiana) según un valor de *threshold*. Como comentamos en el apartado de posibles mejoras, se trata de un algoritmo relativamente simple y que obtendría mejores resultados con patrones parametrizados según el dígito de entrada.

Damos a notar la ausencia de la *feature* “número de transiciones en dirección horizontal” debido a que no nos aporta casi nada de información dado a como son los dígitos que manejamos. La mayoría de ellos tendrían el mismo número de transiciones y por lo tanto hemos decidido no incorporarlo a nuestra lista de *features*.

3.3. Visualización

A partir de los primeros resultados, hemos obtenido los dígitos clasificados erróneamente y los hemos examinado con la función `outputDigit`, que muestra por pantalla la representación del dígito.

De esta manera hemos podido detectar un conjunto de dígitos “anómalos” que suelen acabar siendo mal clasificados, debido a ciertos tipos de caligrafía en cursiva, excesivos detalles en las puntas, errores en el momento de escribir (tachones e intentos de reconducir los trazos correctamente), que en algunas ocasiones pueden hacer difícil el reconocimiento del dígito hasta a un humano, como se puede apreciar en la figura 1.

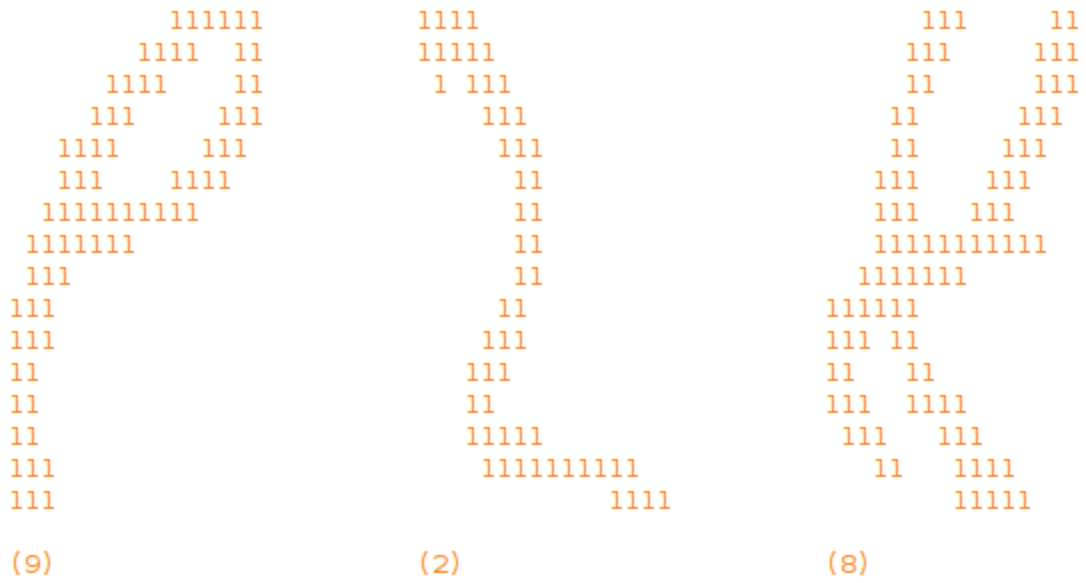


Figura 1: ejemplos de dígitos anómalos, con su valor real entre paréntesis

Hemos preferido no descartar estos casos porque consideramos que pueden darse en situaciones reales de clasificación de dígitos, por lo que no tiene sentido eliminar o modificar estas instancias. Esto también nos ha hecho comprender la dificultad de nuestra tarea, ya que cada persona puede escribir un carácter de una manera única que quizá sea alejada de la forma más habitual, además de los posibles errores humanos durante la escritura.

4. Protocolo de *resampling*

Decidimos utilizar 5x2 *cross-validation* como protocolo de resampling, ya que creemos que nos ofrece un buen compromiso entre el error de predicción a partir de los conjuntos entrenamiento y validación, y el coste computacional. Además, siguiendo las recomendaciones de la descripción del *dataset*, utilizaremos una mitad de los datos para el aprendizaje y la otra para poner a prueba la eficacia del modelo obtenido (*testing*); así, el número de dígitos de cada partición es todavía relativamente grande, y los escogeremos de manera que el número de dígitos de diferentes clases en cada partición sea uniforme.

Para aplicar *resampling* a todos los siguientes experimentos con diferentes modelos vamos a utilizar las rutinas `trainControl` y `train` del paquete `caret`, para simplificar la gestión del protocolo, el proceso iterativo, etc… de manera que trabajemos con un particionado óptimo del conjunto de *training* (que minimice el error de validación).

5. Construcción y ajuste de modelos

5.1. Primeros resultados: métodos lineales

Para empezar a obtener los primeros resultados usaremos LDA usando la función `lda` del paquete `MASS`. Cogeremos una muestra de 796 elementos (mitad del total de dígitos en la muestra) para entrenar al LDA. Empezamos comprobando que con las variables extraídas el error de *training* que es aproximadamente de 0.25%, con esto, el error que obtenemos con el *testing* es del 13.05%. Destacamos que con este modelo, R nos avisa que una o más variables son colineales, y a partir de esto podemos entender que existen variables que son combinaciones lineales de otras entre nuestra selección de *features*; tal como comentamos en el apartado de extensiones, este sería un punto adecuado para implementar un proceso iterativo que descarte variables extraídas que no considere útiles, así reduciendo la dimensionalidad del espacio de *features*.

El segundo modelo que utilizaremos para obtener los primeros resultados es una regresión logística multinomial, mediante la implementación `multinom` del paquete `nnet`.
Empezamos probando su comportamiento con todo el conjunto de variables extraídas (incluída la matriz booleana del dígito), como comentamos en el apartado 3. Vemos que el error de *training* es del 0%, posiblemente indicándonos que se ha producido un sobreajuste. El error de predicción que obtenemos al clasificar los datos de *testing* es de 7.15%, y lo consideramos menor de lo que esperábamos, ya que típicamente un sobreajuste en los datos de *training* suelen ser indicio de haber obtenido un modelo que no generaliza demasiado bien. También hemos intentado aplicar la técnica de regularización para reducir una posible excesiva complejidad del modelo, pero los resultados no muestran ninguna mejora significativa, ni un error de *training* diferente.

5.2. Métodos no lineales

El primer método no lineal que utilizaremos será el *Multilayer Perceptron* (MLP). Hacemos una primera vuelta para encontrar el número de neuronas que minimice el error de predicción, a partir de una selección que hemos hecho con valores altos. En una segunda vuelta aplicaremos regularización mediante el parámetro `decay`, de manera que compensaremos la excesiva complejidad del modelo causada por el alto número de neuronas. Debido a la (relativamente) alta dimensionalidad de nuestros datos y al elevado número de neuronas, notamos que el proceso de modelaje con MLP tiene un alto coste temporal de computación. Finalmente obtenemos que el número óptimo de neuronas es 35, con decay de 0.3. Con estos parámetros conseguimos unos errores del 7.28% y 20.57% de *training* y *testing*, respectivamente.

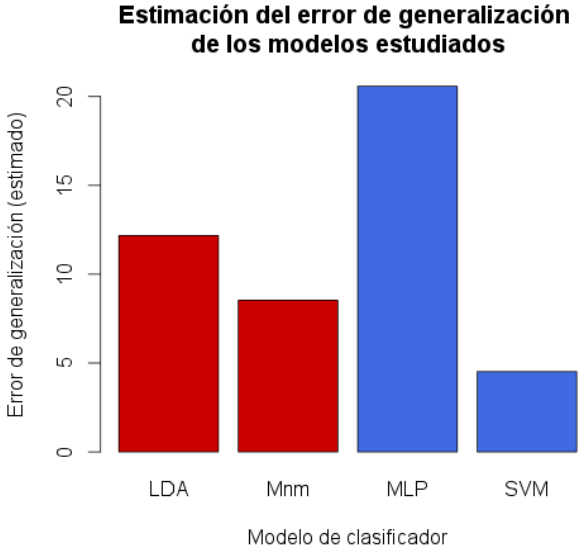
Como segundo método utilizaremos Support Vector Machines (SVM) con kernel Radial Basis Function (RBF), mediante la implementación del paquete `kernlab`. Ésta transformará nuestra tarea de clasificación multiclase en una serie de clasificaciones binarias[1], siguiendo una estrategia *one-versus-one*. En este modelo, el único parámetro que se tiene en cuenta es el coste, ya que el valor de sigma lo obtiene con una estimación automática. El coste óptimo es de 2 unidades, y el valor de sigma encontrado es $1.7983 \cdot 10^{-3}$. Aumentando más el coste paramétrico notamos que el error de entrenamiento es nulo y el de testing empieza a subir, lo que indicaría un problema de *overfitting*; por este motivo decidimos mantener un valor bajo de coste. Se utilizan 654 vectores soporte, algo menor que el número de dígitos de entrenamiento (796), que como comentamos a continuación, afecta al error de *training*.
Con estos parámetros obtenemos unos errores del 0.25% y 3.88% de *training* y *testing*, respectivamente. También hemos obtenido buenos resultados utilizando un kernel lineal, pero RBF es mejor (menor error de *training*/*testing*) por una pequeña diferencia.

En comparación con los primeros resultados, utilizando métodos lineales, no podemos asegurar que la linealidad/no linealidad implique mejores resultados, ya que utilizando el modelos lineales hemos conseguido mejores resultados que con el modelo de red neuronal artificial MLP; sospechamos que reduciendo la dimensionalidad del espacio de *features* podríamos obtener una respuesta más clara. Sí que podemos concluir que el modelaje con métodos no lineales ha sido una tarea más compleja que con los métodos lineales, con la que, experimentando, hemos comprendido la importancia de los parámetros de ajuste y la selección de *features*.

[1] Más detallado en la especificación de `ksvm`: http://www.inside-r.org/node/63499

5.3. Modelo final

Para la elección de nuestro modelo final nos hemos guiado por el error de *testing*, la diferencia entre errores de *training* y *testing* (que puede indicar problemas de infraajuste/ sobreajuste), así como la complejidad del modelo.



Finalmente, a partir de todos los experimentos anteriores, hemos decidido que el mejor modelo para nuestra tarea de clasificación de dígitos manuscritos utiliza SVM con kernel RBF, con los parámetros que explicamos previamente. Hemos podido comprobar su eficacia, además de tener un bajo coste temporal de computación y ser un modelo transparente, que puede ser interpretado a partir de los pesos.

Podemos estimar el error de generalización de nuestro modelo final con el error del modelo SVM sobre el conjunto de *testing*, que es de 3.88% (ver comparación en la figura 2).

Figura 2: estimación del error de generalización; en rojo métodos lineales, en azul no lineales

Dígito \ Modelo	LDA	Mnm	MLP	SVM
0	98.7951	96.3855	87.5000	95.8333
1	90.6666	93.3333	79.2207	100
2	91.7808	93.1506	74.0740	97.5308
3	87.1794	88.4615	77.3809	91.6666
4	80.2325	90.6976	71.7948	93.5897
5	88.1578	84.2105	83.5443	96.2025
6	92.4050	94.9367	86.3636	97.7272
7	79.3103	90.8046	79.2207	94.8051
8	92.3076	89.7435	71.7647	94.1176
9	79.2682	92.6829	84.2105	93.4210

Tabla 1: precisión (%) en la clasificación de dígitos con los diferentes modelos (sobre testing set)

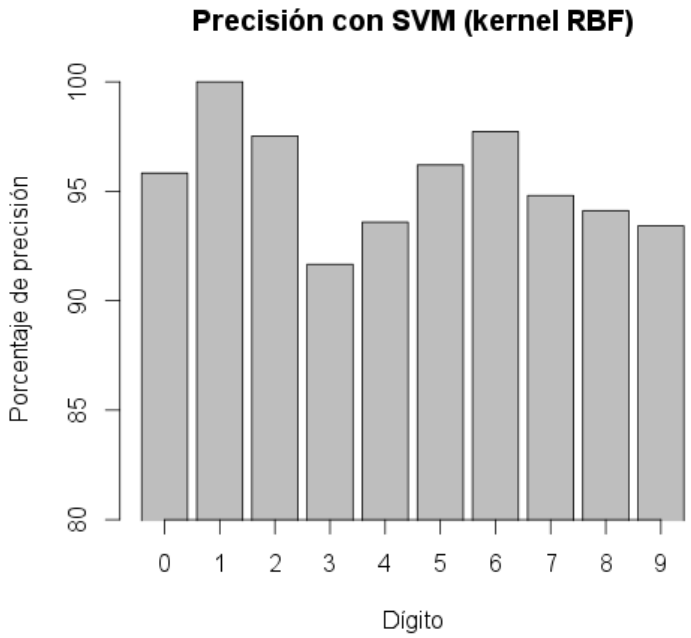


Figura 3: porcentaje de precisión para la clasificación con el modelo final.

Como se puede observar en la figura 3, la precisión de clasificación para todos los dígitos con nuestro modelo final es bastante buena, con mención especial a los casos de 0, 1, 2 y 6, que en diferentes ejecuciones hemos podido comprobar que raramente bajan del 95%. Los dígitos 3, 4 y 9, en cambio, son con los que obtenemos peor precisión, alguna vez incluso bajando del 90%, pero no menos de 80%. Los demás dígitos suelen obtener resultados razonablemente buenos, pero relativamente estables entre varias ejecuciones (normalmente manteniéndose cerca del 95%).

Estas diferencias de precisión suponen una oportunidad para utilizar métodos de comité que, inspirados en la Metanet de Buscema, toman la respuesta de varios modelos para finalmente devolver al usuario una respuesta agregada. Cada modelo podría especializarse en aquellos dígitos donde otros modelos flaquean, de manera que en la respuesta final para la clasificación de un dígito tenga más peso la respuesta de los modelos especializados en éste. Después de estudiar los resultados de todos los modelos que hemos considerado durante varias ejecuciones, creemos que la técnica de *booting* es la más adecuada para nuestra situación, ya que notamos un exceso de *bias* (suponiendo que el comité lo formarán los modelos de esta práctica, aunque seguramente podríamos obtener mejores resultados con una red de SVMs o combinandola con redes neuronales, por ejemplo). En todo caso, por limitaciones de tiempo consideraremos el uso de métodos de comité como una posible extensión a nuestro trabajo.

6. Conclusiones

Tal como detallamos en el apartado anterior, hemos conseguido obtener un modelo que, utilizando SVM, clasifica dígitos manuscritos con una tasa de error baja, y la precisión para todos los dígitos está prácticamente por encima del 90%, por lo que consideramos este experimento un éxito que nos proporciona una base sólida sobre la que podemos seguir trabajando.

Comparando nuestros resultados con los de la MetaNet del *paper* de Buscema, vemos que hemos conseguido resultados parecidos o incluso mejores en el caso de algunos dígitos concretos. Aún así, creemos que no es una comparación justa porque ninguno de los "jueces" de la red es un clasificador con modelo SVM (que ha probado funcionar muy bien con este problema), y además nuestro modelo tampoco utiliza ningún método de comité.

Hemos tenido especial cuidado con la elección de los parámetros de regularización de los modelos para evitar el peligro de sobreajustar, pero sin tampoco caer en el caso contrario. Sin embargo, creemos que podríamos ajustar mejor el modelo MLP, y SVM es posible que sobreajuste los dígitos 1 y 2, pero notamos que disminuyendo el coste empeoramos bastante la precisión de otros dígitos. Esta situación nos ha servido para entender e intentar encontrar un equilibrio en el *trade-off* entre *overfitting* y *underfitting*.

Gracias a la realización de este trabajo hemos podido asimilar y poner en práctica conocimientos vistos en clase como *resampling* con *cross-validation*, regularización de modelos, y sobre todo métodos no lineales como es el caso de nuestro modelo final, con SVM. Esto también nos ha servido para hacernos una idea de la multitud de campos en los que se puede hacer un buen uso de técnicas de *machine learning*. En particular, hemos tenido la oportunidad de atacar un problema de reconocimiento de carácteres (OCR), y en este proceso hemos obtenido una base teórica para introducirnos en este campo que luego hemos podido experimentar, dejándonos ideas como la importancia del paso de *feature extraction*, lo esencial que resulta el preprocesado para futuros resultados, o las diferencias de precisión que podemos obtener con diferentes modelos. También hemos comprendido la complejidad que tiene este tipo de problemas con factores que no podemos controlar, como la caligrafía del escritor, la resolución del dispositivo de entrada, etc. En conclusión, este trabajo nos anima a continuar experimentando e investigando, tanto en este campo concreto como en el mundo del aprendizaje automático.

7. Extensiones, ideas y posibles mejoras

- Nuestro algoritmo para obtener el número de picos/bordes podría ser mejorado añadiendo más patrones parametrizados según la anchura del trazo; actualmente trabaja bajo la suposición que los bordes el dígito acaban en una punta con uno o dos píxeles de anchura.
- Experimentación con métodos de comité. También, de esta manera se podría conseguir una comparación más justa con los resultados de la MetaNet del *paper* de Buscema.
- Implementación de *features* más sofisticadas como número de intersecciones, identificación de curvas, transformadas de Fourier, etc.
- Reducción de la dimensionalidad de nuestro espacio de *features*, posiblemente comportando mejores soluciones con un menor coste computacional.
- Dependiendo del contexto en el que se vaya a utilizar el clasificador, sería conveniente realizar pequeños ajustes en los parámetros del modelo, para balancear el *trade-off* entre *overfitting* y *underfitting*, por ejemplo.

8. Referencias

Buscema, M., & Terzi, S. (2008). Semeion Handwritten Digit Data Set (UCI Machine Learning Repository). Irvine, CA: University of California, School of Information and Computer Science. Retrieved from http://archive.ics.uci.edu/ml/datasets/Semeion+Handwritten+Digit.

Buscema, M, *et al*. (1998). MetaNet: The Theory of Independent Judges. *Substance Use & Misuse, 33*(2), 439-461.

Trier, Ø., Jain, A., & Taxt, T. (1996). Feature extraction methods for character recognition - a survey. *Pattern Recognition*, 641-662.

Frey, P., & Slate, D. (1991). Letter recognition using Holland-style adaptive classifiers. *Machine Learning*, 161-182.

Apuntes de la asignatura *Aprenentatge Automàtic (APA)*, profesor Lluís Belanche.