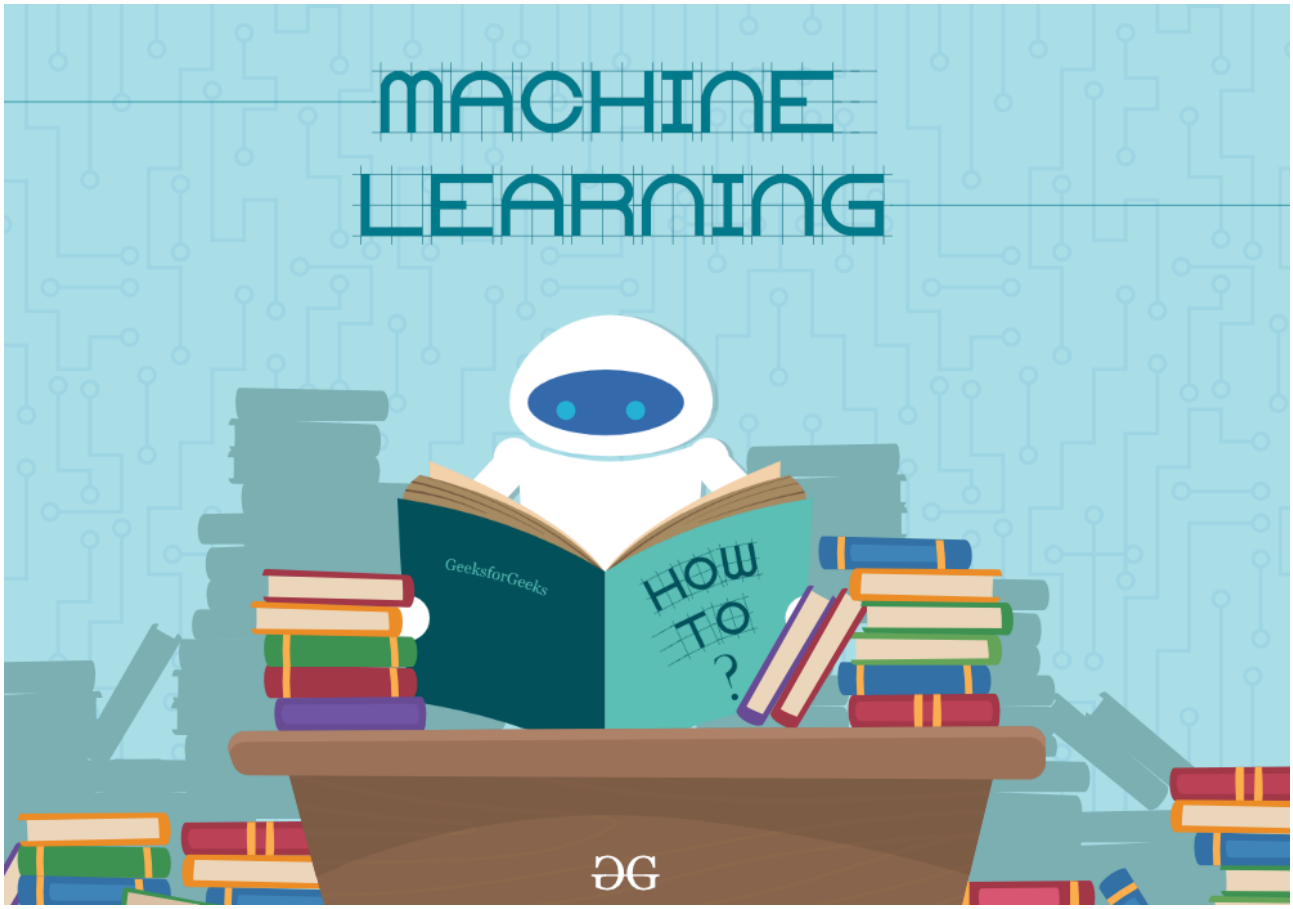
Trabajo Final 2019/20

Introducción al aprendizaje automatizado

LCC

UNR

Franco Sansone



Como primera parte del trabajo se clasificó el dataset de heladas usando svm, naive-bayes con normales y árboles de decisión.

Las implementaciones se entregan adjuntas a este informe, y también se encuentran en mi repositorio personal
https://github.com/francosansone/supportvectormachine/

Para correr svm se usó libsvm, de este link https://github.com/cjlin1/libsvm.
Fue agregada como submodulo al repositorio antes mencionado.

Todo se implementó usando python 2.7.

Entre los archivos se encuentran *generate_folds.py,* el cual debe ejecutarse:
$ generate_folds.py --file path-to-dataset --input number-of-input

path-to-dataset es el path al dataset que se quiere usar, sin la extensión (sin el .data). Debe tener un archivo .names.
input number-of-input es el número de atributos de cada patrón.

Este script genera 10 directorios fold_i con i entre 0 y 9. En cada carpeta se encuentran archivos .test, que representa las 10 particiones que necesitamos para implementar k-folds, junto con el .data que tiene todos los datos restantes. También hay archivos .data.svm y .test.svm con datos análogos a los anteriores pero modificados para la sintaxis de libsvm.

<label> <index1>:<value1> <index2>:<value2> ...
.
.
.

También genera un archivo .nb con lo correspondiente a naive-bayes.

Luego se encuentra un archivo *lib.py* el cual contiene 3 clases, SvmLib, DtreeLib y NaiveBayesLib.

DtreeLib y NaiveBayesLib se usan para entrenanr modelos usando c4.5 y nb_c (implementación del TP3) con .data para entrenar y validar y .test para testing.

Los archivos *train_linear.py* para probar el modelo svm con kernel lineal, *train_polinomial.py* para svm con kernel polinomial, *decision_tree.py* para árboles de decisión y naive_bayes.py para Naive Bayes con Gaussianas.

Por ejemplo, si usamos el dataset de heladas, los mismos deben ser ejecutados así:

$ python naive_bayes.py --file heladas

$ python decision_tree.py --file heladas

$ python train_linear.py --file heladas --iterations 10 --gamma 1.0

$ python train_polinomial.py --file heladas --iterations 10 --gamma 1.0

Para svm los parametros iterations y gamma son optativos.
Los 4 scripts se pueden correr con la flag --debug para ver mensajes de debug durante la ejecución.
Todos escriben en pantalla los resultados obtenidos. La media y desviación estándar de los errores obtenidos para las 10 particiones testeadas.

Un archivo *run_ttest.py* con los errores obtenidos sobre los modelos explicados a continuación, y deuvelve los resultados de ejecutar un t-test en pantalla para el dataset de heladas. El archivo *run_test_bc.py* es análogo pero para el dataset breast cancer.

Una pequeña explicación sobre lo implementado:

SvmLib tiene métodos para ajustar parámetros, entrenar modelos y testearlos en las 10 particiones.

Para ajustar los parámetros primero se utilizó el modelo explicado en la última clase del curso, que consiste en usar una de las particiones para validar y los datos restantes para entrenar.
Primero se generan modelos haciendo saltos grandes entre los valores. Más precisamente, potencias de 10 para c y para gamma también en el caso del kernel polinomial. Para el grado del polinomio se prueba con 2, 6 y 10. Se hacen todas las combinaciones y se guarda la de menor error. Acá deseamos saber en que zona se encuentran los parámetros.

Luego se realiza un segundo ajuste más riguroso. Para ajustar los parámetros basé en lo que aprendimos durante el curso. Mi idea es ir ajustando los parámetros en base a su error. Si el error es grande, dará un salto grande, suponiendo que está lejos del valor ideal. Si el error es pequeño, avanza poco.
El parámetro C se ajusta así:

c = c + c * last_error

Se genera un modelo con ese nuevo parámetro y se testea sobre la partición elegida. Este proceso se repite según el valor con el que corrimos el script (la flag --iterations, por defecto es 10).

Si estamos ejecutando para kernel polinomial, también ajustamos gamma de la misma forma (la flag --gamma, por defecto es 1).
Para ajustar el grado del polinomio, en cambio, hace lo siguiente. Si el valor elegido es 10, se mantiene (números más grandes producían demoras considerables en la ejecución). Si se eligió 2, para cada c y gamma elegidos, se prueba con 2, 3, 4 y 5 (análogo para 6). Guardamos la combinación de parámetros con menor error.
Luego para cada fold, se entrena y testea como para los demás modelos, usando los parámetros antes fijados.

Para complementar esta explicación se pueden ver los comentarios en el código de *lib.py*.

Concluída la explicación de la implementación, prosigo con clasificación de los datasets.

Primero el dataset de heladas. Estos fueron los datos obtenidos:

Modelo	Media del error	Desvío estándar del error
Support Vector Machines - kernel lineal	0.199	0.0677
Support Vector Machines - kernel polinomial	0.209	0.0814
Decision tree	0.231	0.0756
Naive Bayes con Gassianas	0.198	0.061

El mejor resultado fue para Naive Bayes. El ajuste usando Gaussiana es el que mejor funciona para los valores continuos del dataset de heladas. Esto puede deberse a que, en general, si los datos que quiero predecir son parecidos a días en los que hubo helada, hay bastante evidencia de que habrá helada nuevamente.

El peor para Decision tree. Mi conclusión es que los cortes realizados sobre los valores continuos y la poca resistencia al ruido causaron esos resultados.

El kernel lineal funcionó mejor que el polinomial. El ajuste del parámetro C aportó a la resistencia al ruido.

Luego se realizó un t-test con 95% de confidencia y 9 grados de libertad (ejecutamos sobre 10 folds) sobre los modelos Naive-Bayes con Gaussianas (menor error) y Decision Tree (peor error).
Para esto, partimos de la hipótesis de que ambos modelos funcionan igual para este dataset. Para contradecirla, buscamos obtener un valor mayor a 2.26, valor extraído de la tabla 5.6 de la sección 5.6 del libro Machine Learning de Mitchell.

El t-test devolvió 1.6276 para estos dos modelos. Por lo tanto no podemos rechazar la hipótesis. Tampoco podemos concluír que funcionan igual, solo podemos decir que no hay evidencia para decir que uno es peor que el otro.

También se realizó para Naive Bayes y svm kernel lineal, partiendo de la misma hipótesis que el test anterior.

El resultado obtenido fue 2.403. De esta manera, podemos rechazar nuestra hipótesis, concluyendo que, estadísticamente, en el 95% de los casos el método Naive-Bayes va a ofrecer mejores resultados que SVM con kernel lineal.

Finalmente, el trabajo indicaba buscar un dataset que me parezca interesante y aplicarlo sobre svm y algún otro método discutido en el curso.

Tras navegar por la web, elegí un dataset bastante popular sobre cáncer de mama. Básicamente porque me parece interesante cuando la ciencia en general (en este caso, la inteligencia artificial) puede usarse para cosas tan humanas como detectar una enfermedad.

El dataset fue descargado de https://www.kaggle.com/uciml/breast-cancer-wisconsin-data. Consiste en datos extraídos de imágenes digitalizadas. El dataset fue ligeramente modificado para realizar las clasificaciones con nuestros programas. Se eliminó el atributo ID, y se modificaron los valores del atributo diagnosis (M = malignant por 1, B = benign por 0). Los demás

atributos son valores continuos con información útil para el diagnóstico (la cual escapa de mis pobres conocimientos sobre medicina).

El dataset fue clasificado usando SVM con kernel lineal y polinomial y Naive-Bayes.

Se obtuvieron los siguientes resultados:

Modelo	Media del error	Desvío estándar del error
Support Vector Machines - kernel lineal	0.05609	0.0535
Support Vector Machines - kernel polinomial	0.06688	0.05750
Naive Bayes con Gassianas	0.0684	0.0372

Luego ejecuté un t-test con las mismas características que el ejecutado para el dataset de heladas para svm con kernel lineal y Naive Bayes y devolvió el valor de 0.74785. Con 9 grados de libertad. Aún para 90% de confidencia, este valor no nos permite refutar la hipótesis.
También se ejecutó para svm con kernel lineal y kernel polinomia. El test devolvió 0.48168. La conclusión es la misma.