informe

December 7, 2016

Integrantes: Mariela Rajngewerc, Tomás Freilij, Juan Manuel Pérez

```
Reducción de dimensionalidad: Describir brevemente las técnicas empleadas.
Resultados: Describir los resultados conseguidos por los distintos modelos y conjur
Discusión: Analizar los resultados, buscando responder cuestiones como, por ejemplo
```

La longitud sugerida del informe es de entre 3 y 5 páginas de texto (sin contar tablas o figuras). Además pueden incluirse tablas y figuras, pero siempre deben ser referenciadas y explicadas en el texto.

Si se tomaron ideas de la literatura (papers, libros, blogs, wikipedia o lo que sea), citar claramente las fuentes (autor, título, tipo de publicación, año de publicación, URL si corresponde, etc.).

No incluir código. Si es necesario describir un algoritmo, hacerlo en pseudocódigo

```
Populating the interactive namespace from numpy and matplotlib
```

1 Resumen del trabajo

El objetivo de este trabajo fue explorar la performance de distintos clasificadores en la tarea de detección de spam. Se procedió, sobre un corpus de mails etiquetados positiva o negativamente como spam, primeramente a elegir un conjunto de atributos.

Para cada uno de los tipos de clasificadores explorados (`Decision Tree`, `Random Forest`, `KNN`, `Näive Bayes`, y `Support Vector Machine`) optimizamos sus parámetros para mejorar la performance sobre el dataset, a la vez que investigamos cómo afectaba a su rendimiento la utilización de `Principal Component Analysis`.

2 Métodos y Materiales

2.1 Dataset

El dataset constó de 90000 mails, equilibrados entre `spam` y `ham`. De este total de mails, separamos un 10% para test, y usamos el 90% restante como desarrollo.

2.2 Extracción de Atributos

Los atributos principales que se extrajeron de los mails fueron:

- Atributos generados por TF-IDF

- Atributos escogidos manualmente

Nos quedamos con 200 atributos del primero, mientras que del segundo extrajimos 73 adicionales, sumando un total de 273 atributos por mail.

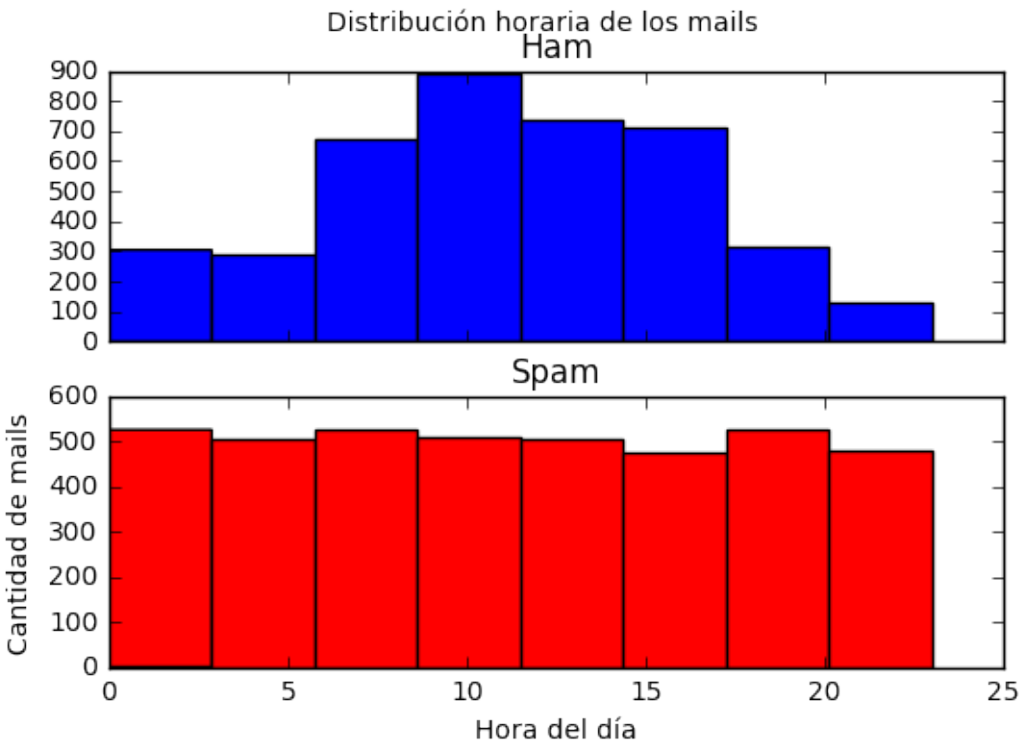
Los atributos extraídos mediante `TF-IDF` fueron realizados sobre el payload del mail y no sobre el texto completo. Para el parsing de los mails en texto plano utilizamos la librería estándar `email`[cita de Python mientras que para la generación de los atributos con `TF-IDF` utilizamos el vectorizador correspondiente de `sklearn`.

Los atributos manuales que extrajimos fueron tanto del header como otros del payload, luego de un análisis observacional del dataset. Del encabezado del mensaje extrajimos:

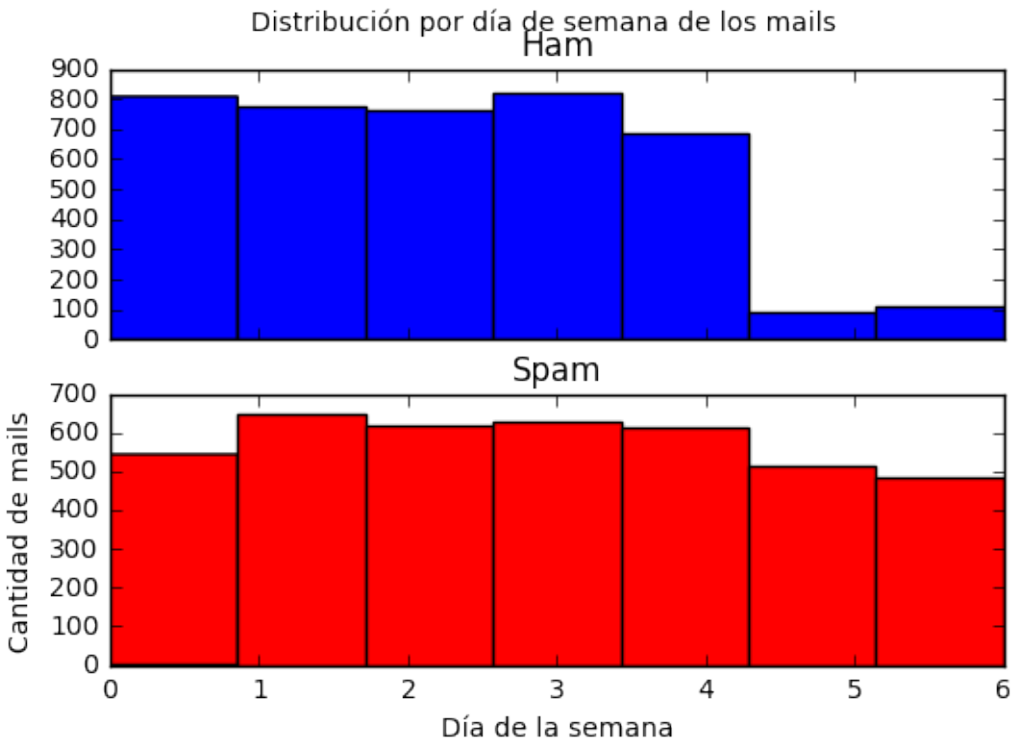
- El `content-type` del mail: si son texto, imágenes, html, etc o bien una mezcla de estos
- La hora en la que fue enviado
- ¿El mail fue enviado entre las 7 y las 20hs?
- El día de la semana (Lunes a Domingo)
- ¿Fue enviado el fin de semana?
- El día del mes (1-31)
- El año en el que fue enviado
- La cantidad de receptores del mail
- ¿El nombre del sender tiene algún caracter no ASCII?

Los atributos temporales del mail fueron extraídos luego de analizar el dataset. Por ejemplo, podemos ver en las figuras siguientes que la distribución horaria del ham tiene casi el 77% concentrado en el rango de 7 a 20 hs, mientras que el spam se reparte uniformemente. También puede observarse como el 95% del ham se envía durante los días de semana, mientras que el spam también se reparte de igual manera en cada día.

```
Frecuencia de Ham entre 5 y 20 hs = 0.77037037037
```



Frecuencia de Ham en día de semana 0.94962962963



2.3 Clasificadores Utilizados

Los tipos de clasificadores explorados en este trabajo fueron los siguientes:

- Decision Tree
- Random Forest
- K-Nearest Neighbors
- Naive Bayes
- Support Vector Machines

Veamos brevemente los hiperparámetros de cada uno de estos modelos.

2.3.1 Decision Trees

La implementación de árboles de decisión en `sklearn` implementa CART. Los hiperparámetros que elegimos optimizar fueron:

- min_samples_split: cuántos elementos tengo que tener como mínimo para partir un nodo
- criterion: gini o entropy, dos medidas distintas de dispersión
- max_depth: profundidad máxima del árbol
- max_features: qué porcentaje de variables (aleatorio) tomo a la hora de partir un nodo

2.3.2 Random Forest

Leyendo la documentación de sklearn respecto a Random Forest, decidimos optimizar los siguientes parámetros:

- min_samples_split y max_features: Idem a Decision Trees
- n_estimators: Cantidad de árboles

Si bien es verdad que cuantos más árboles mejor performance tiene el modelo, también es cierto que a partir de cierto número de estos no mejora significativamente, con lo cual decidimos probarlo también.

2.3.3 K-Nearest Neighbors (KNN)

En el caso de KNN experimentamos con los hiperparametros `n_neighbors`, `weights` y `metric`. `n_neighbors` parametriza la cantidad de consideramos el rango entre 100 y 1000. Dada una nueva instancia el algoritmo devolvera la clase mas ocurrente entre las n_neighbors mas cercanas weights: hace referencia al peso de cada punto, al elegir distance como unica opcion de weight estamos diciendo que los vecinos mas cercanos tienen mas influencia sobre la nueva instancia que los mas lejanos. metric: se refiere a como medir la distancia que mencionamos en el hiperparametro anterior. Dentro de las metricas disponibles en sklearn.neighbors.DistanceMetric decidimos experimentar con "canberra" y "braycurtis".

2.3.4 SVM

Para SVM decidimos exploramos los siguientes conjuntos de hiperparametros: - kernel : lineal o rbf - C: np.arange(0.2, 5.0, 0.4)

En el caso del kernel rbf, consideramos distintas opciones para gamma (entre 0.01, 1.00 con saltos de 0.01) Intentamos estudiar el caso donde el kernel era un polinomio de grado 3 pero luego de 24 horas no habia finalizado asi que decidimos descartar ese caso.

2.3.5 Naive Bayes

Aplicamos los clasificadores Bernoulli y Gaussian, donde los estimadores de ### Optimización de hiperparámetros

Una vez elegidos los conjuntos de hiperparametros donde nos interesaba estudiar cada uno de los clasificadores, nos abocamos a la optimización de estos. Si bien uno en un problema como este tendería a querer optimizar la precisión , nos decantamos por la función que mide el área bajo la curva ROC, que es una medida un poco más amplia.

En lugar de hcer En todos los casos, excepto en SVM, elegimos usar cross validation de 10 folds. En SVM consideramos solo 3 folds ya que el entrenamiento de estos clasificadores es sumamente costoso computacionalmente.

Esta funcion analizaba distintas combinaciones de hiperparametros y nos devolvia la mejor combinacion en cada caso. La eleccion de la mejor combinacion se realizo comparando los valores roc_auc. Es decir,nos quedabamos con la combinacion de hiperparametros que nos daba un valor mayor de area bajo la Curva Roc.

- ROC AUC
- Cross validation (10 vs SVM 3)

- Usamos Randomized (vs Grid)
- PCA (JM)

3 Resultados y Discusión

En primer lugar, el programa hecho consumía demasiada memoria (del orden de los 8 gigabyes), con lo cuál se hizo algo dificultoso el trabajo. Desconocemos si fue un problema de alguna de las librerías, o simplemente fue el tamaño de los datos. La matriz generada no parece ser el problema.

A continuación, mostramos los resultados obtenidos por cada uno de los clasificadores obtenidos (sin aplicar reducción de dimensionalidad)

Clasificador	precision	accuracy	f1	recall	roc_auc
Decision Tree	0.965289	0.967667	0.967749	0.970222	0.967667
Random Forest	0.990007	0.990333	0.990337	0.990667	0.990333
KNN	0.970549	0.968667	0.968604	0.966667	0.968667
Naive Bayes	0.98384	0.832667	0.801686	0.676444	0.832667
Linear SVM	0.900729	0.878556	0.875100	0.850889	0.878556
RBF SVM	0.764366	0.825111	0.843133	0.940000	0.825111

Por otro lado, aplicando previamente PCA:

Clasificador	no.comp	precision	accuracy	f1	recall	roc_auc
Decision Tree	100	0.945659	0.948333	0.948488	0.951333	0.948333
Random Forest	190	0.98361	0.985222	0.985247	0.986889	0.985222
Naive Bayes	220	0.867169	0.846444	0.841985	0.818222	0.846444
KNN						
Linear SVM	260	0.915568	0.829222	0.809423	0.725333	0.829222

En PCA descartamos el kernel RBF ya que produjo resultados más pobres que el kernel lineal.

La comparación de los tiempos es difícil de realizar ya que la optimización no fue igual de exhaustiva en todos los algoritmos. Mientras que en las dos variantes de árboles realizamos 1000 y 100 iteraciones en la búsqueda combinatoria, en SVM hicimos muchas menos. Sí podemos decir que en Random Forest tomó cerca de 66 horas la optimización de sus parámetros (con PCA) y cerca de 35 horas sin PCA.

3.1 Podemos observar que la reducción de dimensionalidad no mejoró sensiblemente ninguno de los clasificadores; de hecho su performance

4 Conclusiones