Universidad de Buenos Aires
Facultad de Ingenieria

Diseño del Trabajo Práctico

Clasificación de Crímenes en San Fransisco

Organización de Datos

Tomás Arjovsky
Vsevolod Gavrilov
Nicolás Ledesma

Los 4 Jinetes De La Entropía
L4JE en Kaggle

24 de septiembre de 2015

Índice

1. Introducción

El trabajo práctico consiste en proponer una solución a la consigna de una competencia abierta en Kaggle, una plataforma de la comunidad de Data Science. La competencia consiste en la predicción de los crímenes ocurridos en San Francisco a partir de los datos proporcionados. Estos datos se dividen en un *set de entrenamiento*, un archivo csv, donde cada fila representa un crímen y cada columna proporciona información acerca de él, y un *set de prueba*. Este último tiene el mismo formato, pero en las filas no se menciona de que crímen se está tratando. Queda a cargo de los participantes el deducir la probabilidad de la ocurrencia de cada *clase* de crímen para cada fila del set de prueba.

Nosotros vamos a encarar el problema de la predicción de las clases de crímen utilizando varios algoritmos de *machine learning*: **Regresión Logística**, **Red Neuronal** (y analizaremos otros algoritmos posibles). Dependiendo de la calidad de la solución de cada algoritmo, juntaremos los mejores *modelos* para tener un resultado más preciso. Necesitaremos procesar y preparar los datos, extraer los diferentes atributos (o *features*) del set de datos para que los algoritmos se entrenen de manera más eficiente. La solución final va a estar escrita en **C++**, pero realizaremos pruebas en lenguajes de más alto nivel (**Python** mayormente), aprovechando las herramientas de ML y procesamiento de datos existentes.

2. Acerca del equipo

2.1. Filosofía de trabajo

Nuestra filosofía para encarar esta competencia fue primeramente reunir toda la información y el conocimiento que creimos necesario para entender el problema. Para esbozar una solución y desarrollar un algoritmo de aprendizaje que consiga hacer una clasificación óptima de los crímenes de San Francisco decidimos primero entender el problema en sus distintas aristas. Esto implicó entre otras cosas:

- Revisar los comentarios hechos hasta el momento en los foros de Kaggle. La competencia lleva ya varios meses de desarrollo, y es una buena forma de hacer un primer acercamiento aprender de los comentarios de los participantes que ya comenzaron a desarrollar su solución.
- Investigar acerca del dominio del problema planteado en la competencia. Las características geográficas de la ciudad de San Francisco, el grado de importancia de los features incluidos en el set de entrenamiento y el set de evaluación. Análisis y comprensión del significado de los features. Por ejemplo, entender qué tipo de eventos comprende la categoría de crimen "Non criminal".
- Compartir los conocimientos previos de los integrantes del equipo en el campo de Data Science, más específicamente en el área de machine learning. Complementar estos conocimientos previos con nuevas investigaciones que nos permitan aprender el funcionamiento de los algoritmos de machine learning existentes, y aún más importante, la idoneidad de los mismos para abordar este problema.

2.2. Plan de trabajo

Desde la conformación del equipo nos dedicamos primeramente a hacer investigaciones con la extensa información que hay en internet, ya sea para aprender los algoritmos, como el dominio del problema y la matemática involucrada.

A continuación, una vez aprendidas las nociones básicas de varios algoritmos y soluciones posibles, pasamos a investigar sobre posibles librerías que nos faciliten el tratamiento de los features de los sets de entrenamiento y evaluación, así como la implementación de algunos de los algoritmos investigados. Así fue como encontramos comodidad en el entorno que ofrece python para rápidamente procesar los datos de entrada y simular, a través de librerías como el Sci-kit, los algoritmos que más nos llamaron la atención y nos parecieron útiles para desarrollar una solución.

Simultáneamente evaluamos y propusimos diversas formas de generar nuevos features a partir de los que aporta el set de entrenamiento. Consideramos que la forma de representar la información otorgada

por la competencia, y de alimentar los algoritmos es clave para lograr un aprendizaje más eficiente en ellos.

Finalmente comenzamos a pensar en la implementación de los algoritmos que más nos gustaron en C++, y las librerías que consideramos útiles para facilitarlo. De aquí en adelante resta comenzar a desarrollar la implementación, y dejar abierta la posibilidad de combinar soluciones para obtener mejores resultados.

3. Tratamiento de datos

Dates	Category	Descript	DayOfWeek	PdDistrict	Resolution	Address	X	Y
2008-04-15 18:30:00	DRUG/NARCOTIC	POSSESSI...	Tuesday	PARK	JUVENILE CITED	700 Block of STANYAN ST	-122.453513	37.768697
2011-08-25 18:33:00	WARRANTS	WARRANT ...	Thursday	SOUTHERN	ARREST, BOOKED	900 Block of MARKET ST	-122.408421	37.783570
2007-09-09 02:32:00	DRUNKENNESS	UNDER IN...	Sunday	SOUTHERN	ARREST, BOOKED	1400 Block of FOLSOM ST	-122.413772	37.772142
2012-09-08 23:59:00	LARCENY/THEFT	GRAND TH...	Saturday	INGLESIDE	NONE	100 Block of EMMETT CT	-122.416079	37.747295
2004-03-17 16:00:00	LARCENY/THEFT	GRAND TH...	Wednesday	SOUTHERN	NONE	CLEMENTINA ST / 5TH ST	-122.403924	37.780699
2014-06-01 00:20:00	BURGLARY	BURGLARY...	Sunday	NORTHERN	NONE	1300 Block of SUTTER ST	-122.422586	37.787387
2008-07-10 18:15:00	OTHER OFFENSES	DRIVERS ...	Thursday	SOUTHERN	ARREST, CITED	HARRISON ST / 1ST ST	-122.392941	37.786068
2003-05-13 12:00:00	LARCENY/THEFT	PETTY TH...	Tuesday	NORTHERN	NONE	0 Block of CERVANTES BL	-122.438303	37.803634
2013-04-07 19:00:00	NON-CRIMINAL	FOUND PR...	Sunday	SOUTHERN	NONE	300 Block of 10TH ST	-122.412410	37.772404
2014-06-25 22:15:00	LARCENY/THEFT	GRAND TH...	Wednesday	SOUTHERN	NONE	FOLSOM ST / 10TH ST	-122.412827	37.772812

Cuadro 1: Algunos datos aleatorios del set de entrenamiento

3.1. Explorando el set de datos

Los sets de datos son extensos. Para darnos una idea de como están compuestos podemos tomar una muestra, y analizarla. Cada fila del set representa un crimen. Las filas vienen con una fecha completa, el distrito donde pasó el crimen, el día de semana, la dirección, la latitud y la longitud exactas. El set de entrenamiento también viene con el tipo de cada crimen (los que justamente necesitamos deducir sobre el set de prueba), y con una descripción y resolución de cada crimen. Estos últimos dos no nos interesan, ya que no aparecen en el set de prueba y, por ende, no podremos tomar ninguna decisión en base a ellos.

Pero para entender mejor el set de datos, lo graficaremos. Teniendo, por ejemplo, el gráfico de categorías 1 ya podemos hacer deducciones sobre nuestro set de entrenamiento. Es muy visible que los crímenes no están uniformemente distribuidos, por ende lo más probable es que en la deducción casi siempre vamos a tender a darle más probabilidad a *LARENCY/THEFT* o *OTHER OFFENSES*.

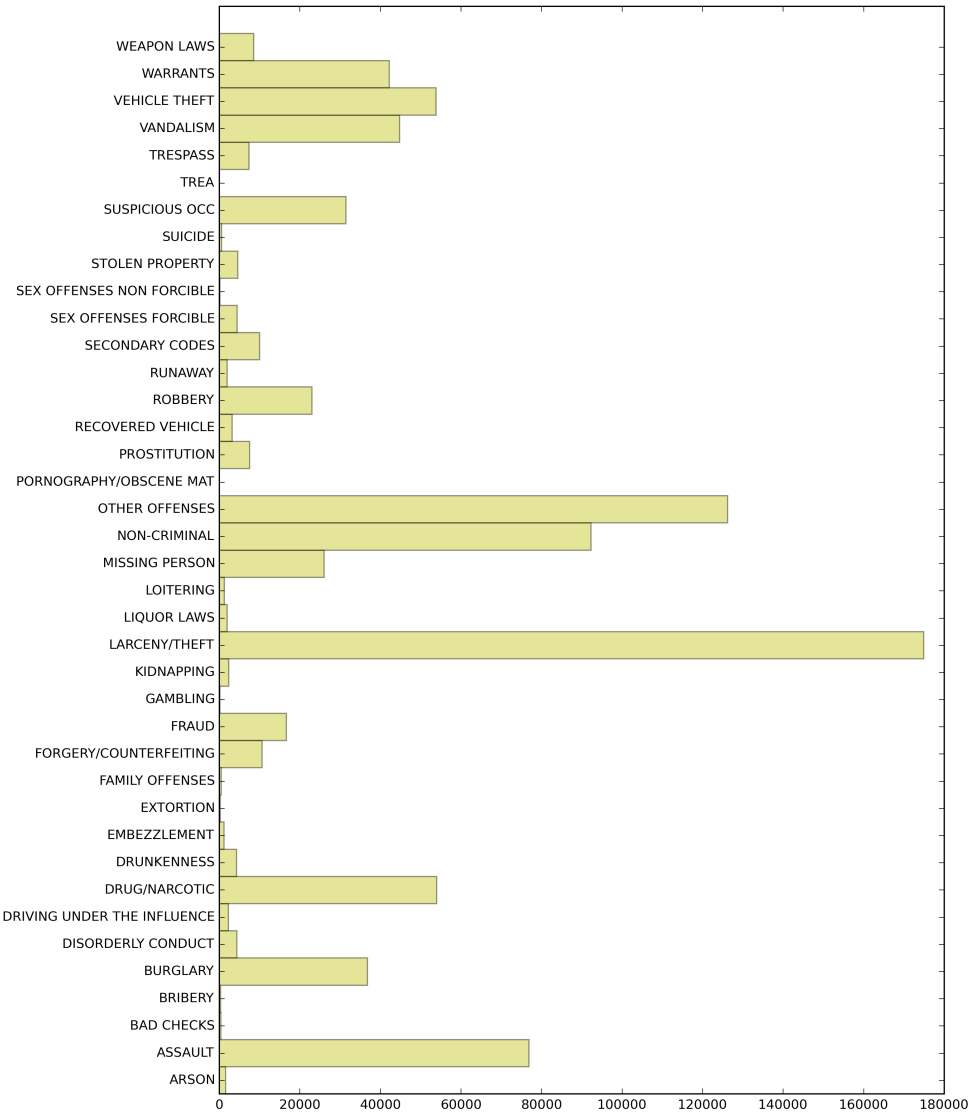


Figura 1: Cantidad de crímenes de cada tipo.

3.2. Deduciendo features

Como parte de preprocesamiento, necesitaremos deducir cuáles son los datos importantes para nosotros, es decir cuáles son los datos que van a ayudar a que nuestro algoritmo aprenda más eficientemente y prediga mejor. Esos datos se pueden encontrar directamente en el set de datos tal como está o pueden necesitar algún procesamiento previo. Algunos inclusive pueden ser combinación de varios datos de una fila.

Los únicos datos discretos, que sirven directamente son el distrito y el día de semana. La fecha tal como viene no tiene valor: cada crimen tiene una fecha única y lo más probable es que en el set de prueba tampoco se repita. Necesitamos procesar esa fecha primero.

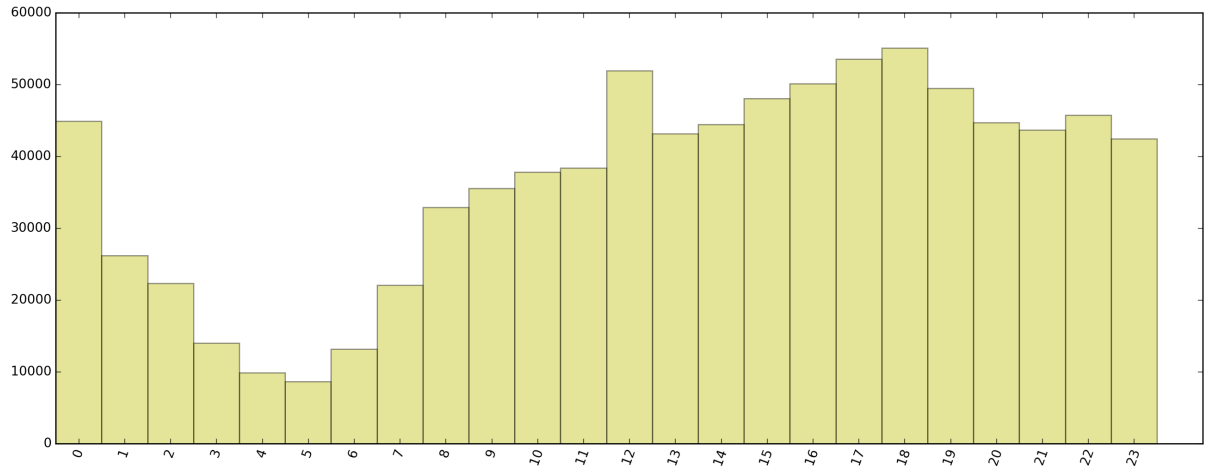
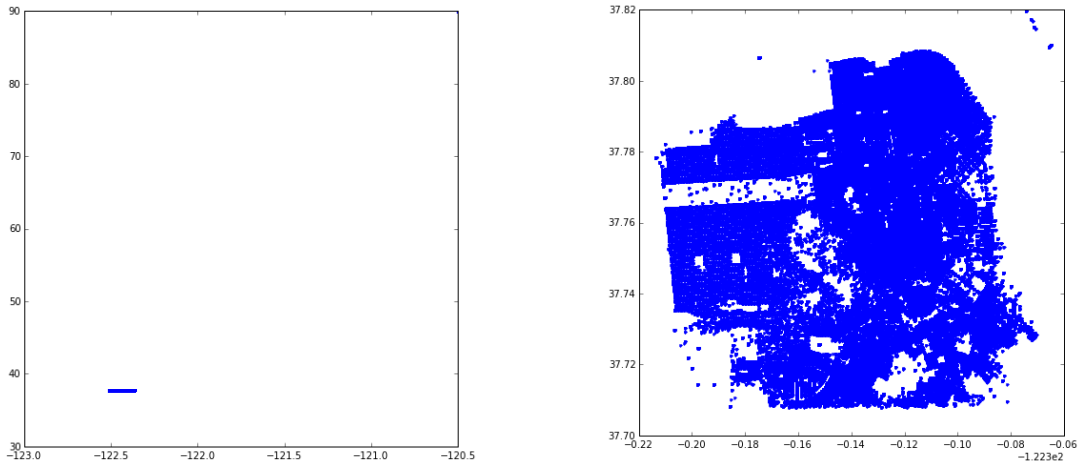


Figura 2: Cantidad de crímenes por hora.

Fecha es un dato muy útil, algunos features resultantes del procesamiento de ella que vienen inmediatamente a la mente son: la hora, el mes, el año. Pero podemos sacar otros features no menores, por ejemplo deducir si era verano o invierno, si era de día o de noche (inclusive variar el rango de día y de noche según estación, siendo que en invierno oscurece más temprano). Sí observamos el gráfico 2, podemos detectar que entre la una y las siete la cantidad de crímenes baja notablemente y, por lo contrario, hay dos picos a las 12 y a las 18. Todos esos datos se pueden convertir en features para nuestro algoritmo. También podemos sacar el dato sobre el día de semana de la fecha - de esta forma la columna con el día de semana en los sets es, en realidad, redundante. Si podemos conseguir datos externos, podríamos que fechas fueron feriado.

La dirección a priori no es un dato tan fuerte como la fecha, y también necesita procesamiento. Podríamos sacar nombres de calles o abusarnos de la notación y deducir que crímenes ocurrieron en las esquinas (las dirección de las esquinas están denotadas como “Calle 1 / Calle 2”).

La longitud y la latitud es a simple vista un dato muy importante y a la vez difícil de procesar. Eso pasa por el hecho de que por separado claramente no tienen sentido. Vamos a tener que encontrar una forma de convertirlos en algo útil para nuestros algoritmos. Una manera podría ser encontrar alguna función que devuelva un número muy parecido para X e Y cercanos. Otra sería dividir las longitudes y latitudes en zonas, y ver en que zona se encuentra la fila. Este enfoque puede conducir a errores ya que para ser preciso va a necesitar más zonas. Algo útil podría resultar una búsqueda de datos externos con latitudes y longitudes de lugares particulares de San Francisco, como pueden ser las estaciones de policía (distancia a ellos podría ser un feature interesante), o determinar si el crimen pasó en un parque.



(a) Set de entrenamiento

(b) Sin datos incorrectos

Figura 3: Gráficos de latitudes y longitudes de los crímenes

Si graficamos la latitud en función de la longitud, lo primero que vamos a obtener es algo parecido al gráfico de 3b: nos encontramos con que algunas de las filas tienen datos incorrectos. Hay 67 crímenes con latitudes y longitudes incorrectas. Si bien podríamos descartarlos - su impacto es minucioso frente al set entero, pero es posible, por ejemplo determinar su X e Y correcto a partir de los crímenes que pasaron en las mismas direcciones. Y para los que queden sin clasificar - podríamos ubicarlos en lugares aleatorios de San Francisco.

3.3. Transformando datos

El set de datos proporcionado en su estado inicial no sirve para aplicarle los algoritmos típicos de machine learning: un algoritmo matemático no tiene una forma práctica de interpretar los datos que contiene. Los datos que mejor puede interpretar son números. Entonces está entre nuestras necesidades el convertir los sets de datos heterogéneos en matrices numéricas.

Todos los features necesitan ser escalados a un rango de números (puede ser entre 0 y 1, o entre -1 y 1). De esta manera el algoritmo tiene menor chance de equivocarse dándole mayor peso a features con mayor valor numérico (como puede ser el año con respecto al mes).

¿Pero como hacemos para representar numéricamente si un crimen pertenece a un distrito? Una de las solución es utilizar codificación *one-hot* (o variables *dummy*). Este método consiste en, en vez de representar alguna variable categórica, como puede ser el distrito, con un solo feature, representarlo con un feature por cada categoría existente, marcando si es o no es de esa categoría (con un 1 o un 0). De esa manera a partir de la columna de Distrito tendríamos 10 features diferentes, donde cada fila tendría un 1 en uno de ellos - y 0 en el resto. Lo mismo es posible hacerlo con los días de semana, meses, años, zonas de latitudes y longitudes, y hasta calles - pero es necesario hacer un análisis previo y decidir si tiene sentido o no.

4. Algoritmos de predicción

4.1. Regresión Logística

La regresión logística a pesar de su nombre es un algoritmo de clasificación, el cual puede implementarse para realizar una clasificación binaria, o una clasificación de múltiples valores posibles.

El modelo de la regresión logística es análogo al de la regresión lineal, dado que tal como la misma,

parte de una función hipótesis que se le aplica a los datos de entrada.

$$h_\theta(x) = \theta^T \cdot x \tag{1}$$

Esta es la representación vectorial de la función hipótesis de la regresión lineal. Donde X es el vector traspuesto de datos x_i, y Θ^T es el vector de pesos θ_j.

En el caso de la regresión lineal, el valor de retorno de la función hipótesis puede variar mucho dependiendo del vector de datos de entrada X y el vector de pesos Θ. En este aspecto la regresión logística se diferencia de la regresión lineal.

4.1.1. Función sigmoidal.

En el caso particular de la regresión logística, la función hipótesis se construye de la siguiente manera:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T \cdot x}} \tag{2}$$

Similarmente, la hipótesis de la regresión logística utiliza el vector de datos de entrada X y el de pesos Θ, y su función retorna un valor escalar. Sin embargo, esta nueva función otorga una ventaja al retornar un número escalar real comprendido entre 1 y 0. Este tipo de función se llama función sigmoidal, y el valor escalar comprendido entre 1 y 0 se puede interpretar como una probabilidad.

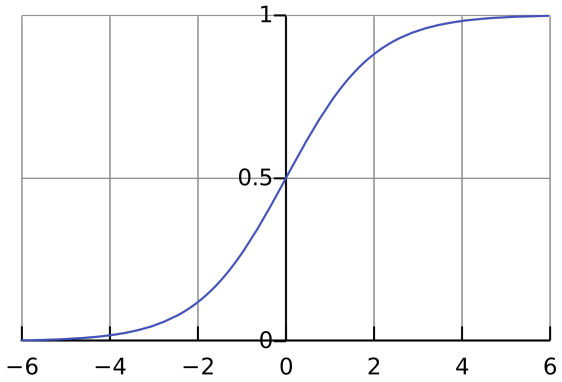


Figura 4: Gráfica de una función sigmoidal.

4.1.2. Uno contra todos.

Sin embargo, hasta el momento el algoritmo parece ser sólo diseñado para resolver problemas de clasificación binaria. Para utilizar este algoritmo de forma pueda abordar un problema de salida multiclase, debe aplicarse algún esquema de clasificación que aproveche las salidas binarias (o de probabilidad entre 0 y 1) de la función sigmoidal. Para eso se utiliza el esquema *ovr (One-vs-rest)*. One-vs-rest no aplica otra cosa que el criterio de *dividir y conquistar*. Por cada clase se resuelve el problema en forma binaria, donde el caso positivo (1) corresponde a la clase misma, y el caso negativo corresponde a cualquier otra clase. A continuación puede verse una representación gráfica de esta estrategia:

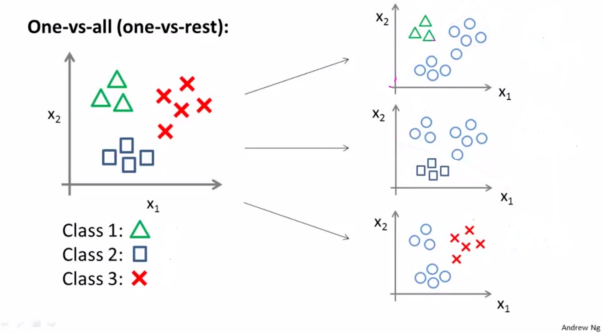


Figura 5: Representación gráfica one-vs-rest. [3]

4.1.3. Estimación y Función de Costo.

De esta manera se cuenta con una serie de funciones $h_\theta^k(x)$ donde k es el número de la clase que representa la función. A través del valor de retorno de todas estas funciones se obtiene entonces la probabilidad de cada clase distinta para cada caso del set de valores de entrada.

Este conjunto de probabilidades obtenido, es una estimación, y esa estimación debe aproximarse al valor deseado. En este trabajo disponemos de un set de entrenamiento asociado a las clasificaciones correctas. El objetivo de la regresión logística es esbozar una función de costo dados los pesos θ_j . De esta manera, optimizando (o minimizando) la función costo se pueden corregir los pesos θ_j, de forma que las estimaciones posteriores se asemejen cada vez más a los valores deseados o "correctos".

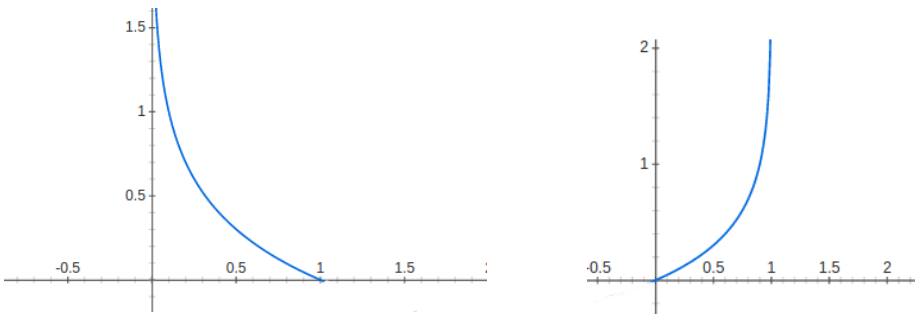
A continuación se muestra un ejemplo ilustrativo de una definición de función de costo:

$$J(h_\theta(x), y) = \begin{cases} -\log h_\theta(x) & si\ y \equiv 1 \\ -\log(1 - h_\theta(x)) & si\ y \equiv 0 \end{cases} \tag{3}$$

La conveniencia de esta función de costo es que:

- Cuando el valor esperado es 1, y la estimación se aproxima a 0, el costo tiende a ∞.
- Cuando el valor esperado es 1, y la estimación se aproxima a 1, el costo tiende a 0.
- Cuando el valor esperado es 0, y la estimación se aproxima a 1, el costo tiende a ∞.
- Cuando el valor esperado es 0, y la estimación se aproxima a 0, el costo tiende a 0.

Esto ocurre gracias a que las funciones logarítmicas de cada caso se comportan como describen las siguientes gráficas:



(a) $y = -\log(x)$ (b) $y = -\log(1 - x)$

Figura 6: Gráficas

Análogamente, se pueden definir funciones de costo de forma contínua, obteniendo comportamientos similares. En el caso particular de la clase LogisticRegression implementada en el paquete de python *sklearn*, una función de costo utilizada es la siguiente:

$$J(\theta, c) = \theta^T \cdot \theta + C \sum_{i=1}^{n} \log(e^{-y_i \cdot (X_i^T \theta + c)}) \tag{4}$$

Donde:

- X_i es el vector de parámetros de entrada del i-ésimo caso de un set de entrenamiento.
- y_i es el valor correcto conocido del i-ésimo caso de un set de entrenamiento.
- c es una constante de ajuste, al igual que los pesos comprendidos en el vector θ.
- $\theta^T \cdot \theta$ y C son complementos de la función de costo pensados para regularizarla, suavizando su comportamiento con el fin de evitar cometer *overfitting*.

4.1.4. Optimización: descenso por el gradiente.

Por último, queda detallar el método de optimización utilizado por la regresión logística. Los métodos de optimización de la función de costo pueden ser derivativos o no derivativos.

El método derivativo clásico es el descenso por el gradiente. El descenso por el gradiente consiste en calcular el gradiente de la función costo. De esta forma se obtiene la dirección y sentido de máximo crecimiento de la función. Con el fin de minimizarla, lo que se hace es, a través de un escalar que se denomina *paso* de la optimización, se van variando los valores de θ_i en la misma dirección y sentido opuesto que el gradiente, de forma de apuntar hacia la minimización de la función de costo. Esto se realiza iterativamente, recalculando el gradiente de la función costo con los parámetros θ_i actualizados.

Sin embargo, las librería *sklearn* utiliza en el modo *liblinear* un método no derivativo denominado *descenso coordenado* [4]. El descenso coordenado consiste en partir la optimización en optimizaciones más simples, de una variable. Esto lo hace minimizando la función de costo por una coordenada o variable a la vez. Itera por cada variable y calcula la dirección de descenso de la función modificando el valor de esa sola variable.

A continuación se muestra una gráfica que ilustra este método de optimización:

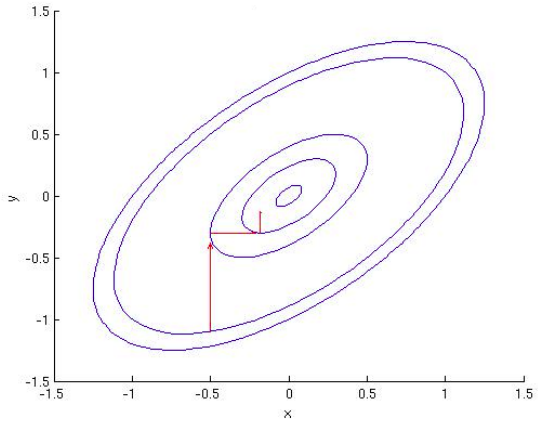


Figura 7: Descenso coordenado. Las elipses concéntricas son curvas de nivel, donde el centro de las elipses se encuentra un mínimo local. [5]

En su forma vectorial, el descenso por el gradiente podría verse con la siguiente fórmula de actualización:

$$\theta := \theta - \alpha \nabla J(\theta) \tag{5}$$

4.1.5. Feature Scaling

Una mejora muy grande que se le puede hacer al descenso por el gradiente es la normalización o escalado de las variables de entrada (features). El problema que ataca es la diferencia entre los rangos de valores posibles de las features.

Por ejemplo, una variable puede ser la cantidad de cuartos que tiene una casa y otro la cantidad de metros cuadrados que tiene. Mientras uno se mueve entre 1 y 10 aproximadamente, el otro seguro será 10 veces más grande. Otros más comunes son, por ejemplo, la calificación del 1 al 5 de una película contra la cantidad de vistas que tiene, que pueden ser millones. Este desequilibrio produce, en gráficos como la figura 7, que se haga cada vez más ovalado, y por lo tanto el paso α que tiene que dar en cada iteración para que no haya divergencia tiene que ser menor. Esto hace que sea más difícil de debuggear y que la convergencia, debido al paso más chico, sea mucho más lenta.

El feature scaling consiste en pasar todas las variables a intervalos aproximadamente entre 0 y 1 y luego entrenar θ sobre esas nuevas features normalizadas. El scaling se hace de la siguiente manera:

$$x_i := \frac{x_i - \mu_{x_i}}{\sigma_{x_i}} \tag{6}$$

Al quitarle el valor medio y dividir por la desviación estandar, todas las variables quedan en rangos similares y el descenso por el gradiente puede hacerse de forma más segura y rápida.

Para poner un ejemplo concreto, probando un ajuste de parábolas por regresión lineal, sin escalamiento de variables requería un α más pequeño que 0.0001 para converger y por lo tanto miles de iteraciones. Por otro lado, escalando las variables, había convergencia con un paso de 1.0 y en algunas decenas de iteraciones ya se conseguían errores menores a 10^{-15}.

4.1.6. Regularización

Como fue mencionada antes, la regularización es una medida que se toma en contra del overfitting. El overfitting, o sobreajuste, es un problema que consiste en que el algoritmo predictor quede muy bien entrenado para el predecir dentro del set de entrenamiento, pero no tenga capacidad de generalizar. Esto suele ocurrir cuando se modela un problema con una complejidad mayor a la que le corresponde, o cuando faltan ejemplos para la cantidad de features que se proponen.

A veces quitar features no es una opción. Sin embargo, si se puede penalizar darle mucha importancia a todos los features a la vez. Esta penalización se hace a través del costo y se llama regularización:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}(log(h_\theta(x^{(i)}))) + (1-y^{(i)})log(1-h_\theta(x^{(i)}))\right] + \frac{1}{m}\sum_{j=1}^{n}\theta_j^2 \tag{7}$$

Y en forma vectorial:

$$J(\theta) = \frac{1}{m}\left[-y^T log(h_\theta(x)) - (1-y)^T log(1-h_\theta(x)) + \theta^T\theta\right] \tag{8}$$

De todos modos hay que tener en cuenta que no se regulariza el parámetro del término independiente, con lo cual en el término de la derecha ($\theta^t\theta$) deben extraerse primero los θ_0.

4.1.7. Stochastic Gradient Descent

Para conjuntos de datos muy grandes, tales como el que se nos presenta de crímenes en San Francisco, puede ser muy costoso sumar sobre todos los 800.000 ejemplos antes de avanzar un paso. El descenso por el gradiente estocástico, mezcla aleatoriamente los ejemplos y luego va iterando sobre aquellos, en cada paso bajando por la contribución que tiene cada uno al gradiente.

El descenso por el gradiente clásico (en tanda), según lo expuesto en la ecuación 5, puede expresarse componente a componente:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \tag{9}$$

Pero podemos observar que las componentes de la sumatoria son la parte que cada ejemplo contribuye al gradiente. Tomando este concepto, en lugar de sumar todos los valores antes de avanzar, el descenso por el gradiente estocástico baja un poco por cada ejemplo que recorre, a través de la contribución que este otorga al gradiente [6].

$$\theta_j := \theta_j - \alpha (h_\theta(x^{(i)} - y^{(i)})) x_j^{(i)} \tag{10}$$

Esto último se hace para cada uno de los ejemplos y luego se vuelve a comenzar con el primero, hasta que el resultado sea el deseado.

Una variante de este método es una mezcla entre el descenso por el gradiente en tanda y estocástico, que agarra pequeñas tandas, de por ejemplo, 10 casos de entrenamiento cada una. Esto permite aprovechar las ventajas de vectorizar las cuentas que provee la versión en tanda y la ventaja de no tardar tanto tiempo hasta hacer un paso que provee la implementación estocástica.

Este algoritmo será útil también en las redes neuronales, que utilizan descenso por el gradiente.

4.1.8. Entrega en Kaggle

En particular, con una implementación rudimentaria en python utilizando solo el distrito y features basados en la fecha, sin ninguna de todas las mejoras aquí planteadas, se obtuvo un puntaje de 2.59489.

Aplicando este algoritmo sobre un subset de entrenamiento como prueba se obtenían predicciones de precisión muy similar al set de entrenamiento, que rondaba el 22 % de acertadas.

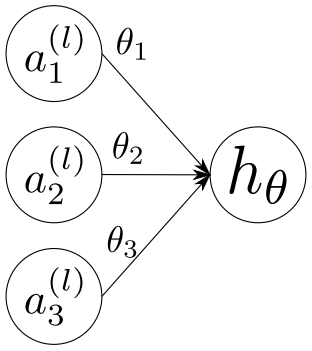
4.2. Redes Neuronales

Las redes neuronales son estructuras de predicción de propósito general, que se basan en unidades, llamadas *neuronas*. Particularmente en este caso, vamos a usar una red neuronal para clasificación, y veremos como influye eso a la hora de construirla.

Históricamente surgieron de un acercamiento al aprendizaje automático que seguía como heurística conseguir un parecido entre la inteligencia artificial y el funcionamiento de la mente humana. Una red tiene capas de neuronas, cada una de las cuales obtiene con input una combinación lineal de los datos de la capa anterior. Cada neurona, a ese input, le aplica una *función de activación*, que será su output.

Para el caso de una sola neurona de clasificación, podemos comparar el comportamiento con el de una regresión logística:

Figura 8: Neurona clasificadora



En la figura se ve una capa l (de "layer") con tres neuronas. Cada una tiene como output su función de activación a. La neurona de la capa siguiente toma como input una combinación lineal de las anteriores y le aplica su propia función de activación, la función sigmoidea.

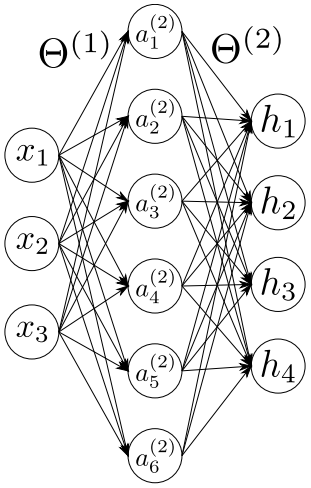
$$h_\theta = g(z) \tag{11}$$

$$z = \sum_{i=1}^{3} \theta_i a_i^{(l)} = \theta^T a^{(l)} \tag{12}$$

De esta forma obtenemos la ya vista regresión logística.

Una red neuronal es más general. Tiene una capa inicial, con los datos de entrada para la predicción, una capa de salida, con la hipótesis, y una cantidad arbitraria de capas llamadas *ocultas* entre ellas. Estas últimas permiten que la hipótesis final sirva para modelar problemas muy complejos, ya que las activaciones de cada capa anterior actúan como nuevas features para las capas siguientes:

Figura 9: Red Neuronal



La red presentada en la figura, por ejemplo, tiene 3 capas. La de las features del ejemplo (input), una capa oculta (capa 2) y una capa de output, que es la que da como resultado la hipótesis. La necesidad de múltiples hipótesis corresponde a que la clasificación de crímenes es multiclase. Cada neurona del output dará como resultado la probabilidad de pertenencia del ejemplo a cada clase.

4.2.1. FeedForward

El paso hacia adelante (obtención de la hipótesis según features) se hace repitiendo el esquema de la neurona única. La diferencia radica principalmente en que ahora en lugar de tener un vector θ, como de las salidas de un una capa de neuronas obtenemos las entradas de la siguiente (de un vector obtenemos otro), necesitaremos en cada paso una matriz que llamaremos $\Theta^{(l)}$.

Las capas se relacionan de la siguiente manera:

$$\text{Activación:} \quad a^{(l)} = g(z(l)) \tag{13}$$

$$\text{Combinacion lineal:} \quad z^{(l+1)} = \Theta^{(l)} a^{(l)} \tag{14}$$

Y particularmente para la red de arriba se cumple que

$$a^{(1)} = x \tag{15}$$

$$a^{(3)} = h_\Theta \tag{16}$$

4.2.2. Función de costo

Como es un problema de clasificación, al igual que en un problema de regresión logística, se usa una función de costo logarítmica, pero se suma el costo para la probabilidad asignada a cada clase.

$$J(\Theta) = -1/m \left[\sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} log(h_\Theta(x^{(i)})_k) + (1 - y_k^{(i)})(1 - log(h_\Theta(x^{(i)})_k)) \right] \tag{17}$$

siendo m la cantidad de ejemplos y K la cantidad de clases.

Una observación importante es que el costo se puede obtener como un promedio de los costos individuales de cada ejemplo:

$$J(\Theta) = 1/m \sum_{i=1}^{m} J(\Theta)^{(i)} \tag{18}$$

Una segunda observación es que basta conocer la salida de la última capa para poder obtener el costo de la predicción. Estas dos serán asunciones que tendremos que hacer más adelante para el algoritmo de *Backpropagation*.

4.2.3. Backpropagation

Como se desea hacer descenso por el gradiente para minimizar el costo, es necesario computar el gradiente de la función de costo según los parámetros $\Theta_{ij}^{(l)}$. Sin embargo, calcular manualmente las derivadas de la hipótesis en cuanto a los parámetros de todas las matrices se vuelve complicado por la cantidad de pasos intermedios que hay entre la primera y la última capa. Además, si así se calculara sería poco generalizable ya que dependería de la estructura de la red y de la cantidad de capas ocultas.

Para solucionar este problema surgió en los 70[2] un algoritmo llamado *backpropagation*, que permite calcular las derivadas parciales en función capa a capa, partiendo de los resultados. Recordando la primera observación sobre la función de costo, trabajaremos solo con el costo individual para un ejemplo, y para simplificar la notación, lo llamaremos simplemente J.

Para comenzar, definimos provisionalmente un error de cada neurona como el cambio del costo en función de su entrada:

$$\delta_i^{(l)} = \frac{\partial J}{\partial z_i^{(l)}} \tag{19}$$

Particularizamos este error en la última capa (L):

$$\delta_i^{(L)} = \frac{\partial J}{\partial z_i^{(L)}} = \frac{\partial J}{\partial h_{\Theta_i}} \frac{dh_{\Theta_i}}{dz_i^{(L)}} = \frac{\partial J}{\partial h_{\Theta_i}} g'(z_i^{(L)}) \tag{20}$$

Para el caso particular de la activación sigmoidea y del costo logarítmico:

$$\delta_i^L = h_{\Theta_i} - y_i \tag{21}$$

O directamente en forma vectorial:

$$\delta^L = h_\Theta - y \tag{22}$$

Teniendo este error, lo que backpropagation propone (como su nombre lo indica) es propagar el error hacia atrás, obteniendo el error de la capa l en función del de la capa $l+1$. Llamo J_l al costo en función de los inputs z de la capa l (o sea, $J(z_1^{(l)}, ..., z_{n_l}^{(l)})$), y J_{l+1} al costo en función de los costos de la capa l+1(o sea, $J(z_1^{(l+1)}, ..., z_{n_{l+1}}^{(l+1)})$). De esta forma, podemos decir:

$$J_l(z^{(l)}) = J_{l+1}(\Theta^{(l)} g(z^{(l)})) \tag{23}$$

y aplicando la regla de la cadena en forma vectorial:

$$\nabla J_l(z^{(l)})^T = \nabla J_{l+1}(\Theta^{(l)} g(z^{(l)}))^T D(\Theta^{(l)} g(z^{(l)})) \tag{24}$$

Los gradientes J_l y $J_l + 1$ son las derivadas de la función de costo respecto a $z^{(l)}$ y $z^{(l+1)}$, que son justamente los errores δ de aquellas capas. La matriz diferencial de la derecha se calcula fácilmente y la ecuación anterior queda:

$$\delta^{(l)^T} = \delta^{(l+1)^T}\Theta^{(l)} \odot g'(z^{(l)})^T \tag{25}$$

Donde $\odot$ es el producto elemento a elemento entre vectores, o "Producto de Hadamard". Trasponiendo:

$$\delta^{(l)} = \Theta^{(l)^T}\delta^{(l+1)} \odot g'(z^{(l)}) \tag{26}$$

Con lo cual obtuvimos el error en una capa, vectorialmente, dado el error de la capa siguiente. Particularmente, como la función de activación g es la sigmoidea y su derivada es conocida:

$$\delta^{(l)} = \Theta^{(l)^T}\delta^{(l+1)} \odot g(z^{(l)}) \odot (1 - g(z^{(l)})) \tag{27}$$

$$\delta^{(l)} = \Theta^{(l)^T}\delta^{(l+1)} \odot a^{(l)} \odot (1 - a^{(l)}) \tag{28}$$

Ahora, lo único que falta es calcular, teniendo los errores δ de cada capa, poder calcular las derivadas del costo no según las entradas, sino según los parámetros de cada Θ, que era nuestro objetivo original. De forma análoga a la última resolución, llamamos $J_{\Theta^{(l)}}$ al costo como función de los parámetros de combinación lineal de la capa l a la capa $l + 1$ y $J_{z^{(l+1)}}$ al costo como función de las entradas de la capa $l + 1$.

$$DJ^T_{\Theta^{(l)}} = \nabla J^T_{z^{(l+1)}} D_{\Theta^{(l)}}(\Theta^{(l)} z^{(l)}) \tag{29}$$

Como lo que nos interesa no es en sí la forma vectorial separada por capas, sino armar un gradiente con todos los parámetros, entonces queda expresado cada componente de la siguiente forma:

$$\frac{\partial J}{\partial \Theta^{(l)}_{ij}} = \frac{\partial J}{\partial z^{l+1}_j} g(z^{(l)}_i) = \delta^{(l+1)}_j a^{(l)}_i \tag{30}$$

De esta manera pudimos, a través de únicamente lo obtenido en la última capa (hipótesis) generar las derivadas parciales respecto a cada parámetro para un ejemplo en particular.

Resumiendo los pasos:

FeedForward: se toma el ejemplo y se obtiene la hipótesis yendo hacia adelante en las capas.

Backpropagation: se toma la hipótesis obtenida en el paso anterior y se buscan las derivadas según los siguientes pasos:

1. Calcular el error de la última capa como diferencia entre la hipótesis y los datos conocidos (ecuación 22).
2. Propagar los errores hacia atrás con la matriz de parámetros traspuesta y la capa siguiente (ecuación 28).
3. Una vez obtenidos los errores calcular a través de ellos las derivadas parciales respecto a los parámetros con la ecuación 30.

4.2.4. Bias

Al igual que en la regresión logística se añadía un término independiente a la combinación lineal, en las redes neuronales se añade en cada capa una neurona que tiene siempre activación (salida) 1. Esto no afecta las cuentas hechas atrás. Lo único que cambia es el tamaño de las matrices Θ, que tendrán una columna más para cada capa.

4.2.5. Regularización

También, como ya veníamos haciendo, para evitar el overfitting, habrá una penalización en el costo para los valores de parámetros muy altos. El costo con regularización es el siguiente:

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(h_\Theta(x^{(i)})_k) + (1-y_k^{(i)})(1-log(h_\Theta(x^{(i)})_k))\right] + \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{j=1}^{s_l}\sum_{i=1}^{s_{l+1}}(\Theta_{ij}^{(l)})^2 \quad (31)$$

Como el término que se agrega está sumado al costo anterior, el backpropagation planteado sirve, en tanto y en cuando se le agregue la derivada del costo por regularización. Algo a destacar es que los índices parten del 1 y no del 0 para i ya que no se regularizan los parámetros de bias. Quedará entonces:

$$\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l)}} = \frac{1}{m}\sum_{ej=1}^{m}\frac{\partial J_{BP}^{(ej)}}{\partial \Theta_{ij}^{(l)}} + \frac{\lambda}{m}\Theta_{ij}^{(l)} \qquad \text{para} \qquad j \neq 0 \quad (32)$$

$$\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l)}} = \frac{1}{m}\sum_{ej=1}^{m}\frac{\partial J_{BP}^{(ej)}}{\partial \Theta_{ij}^{(l)}} \qquad \text{para} \qquad j = 0 \quad (33)$$

Siendo $\frac{\partial J_{BP}^{(ej)}}{\partial \Theta_{ij}^{(l)}}$ la derivada parcial individual del costo para el ejemplo ej respecto del parámetro correspondiente utilizando backpropagation.

4.2.6. Método de ajuste

Para la red en sí, la arquitectura inicial pensada es una capa oculta, con 2 o 3 veces la cantidad de nodos respecto de la capa de input. Después de lograr estabilizar el λ de la regularización y el α del descenso por el gradiente, el plan es agregar una o dos capas ocultas más de la misma cantidad de nodos y evaluar el desempeño en el cross validation set (la parte separada del training set para hacer pruebas).

Para asegurarse de que las derivadas están siendo correctamente calculadas por el backpropagation se utilizará gradient-checkin, un método para obtener una aproximación numérica del gradiente para tener algo con qué comparar. Si bien no será una buena aproximación, diferencias muy grandes, como cambios de signo u órdenes de magnitud, indican que errores de implementación del backpropagation.

4.3. Otros Algoritmos

4.3.1. Random Forest

Este algoritmo, conocido en castellano como "Selvas Aleatorias" es un algoritmo de ensamblado que combina varios algoritmos *debiles* (por si solos no logran una clasificación eficiente) para construir un resultado mejor. Random Forest consta de varios arboles de predicción que se entrenan de forma independiente, cada uno con un subset de features de los datos de entrada - de esta manera se obtiene una colección de arboles no correlacionados. Para obtener la predicción final, los resultados de cada arbol se promedian. Para obtener resultados se le pueden variar la cantidad de arboles a usar, la cantidad de features por arbol (suele ser bastante más chica que la cantidad de features total, como la raiz cuadrada de ella, por ejemplo) y la profundidad de cada arbol.

El concepto es interesante, pero su aplicación con nuestro set falló en la práctica. Hemos hecho pruebas con las implementaciones de Random Forest de **scikit**. Dió excelentes resultados sobre el set de entrenamiento, primero entrenado, y luego con las predicciones sobre subsets de los datos de entrenamiento se obtiene un score mayor a 90 % y un *logloss* de $\tilde{0}.4$. Pero se equivoca mucho sobre los datos que no conoce - al entrenarlo con un subset de entrenamiento y probarlo con otro subset, el score se vuelve muy bajo y el *logloss* altisimo. La submisión a Kaggle nos resultó en un *logloss* de 19.25726, usando como features el distrito, los días de semana, la hora, el mes y el año. Cabe destacar que solo lo hemos probado

1 vez, con 10 arboles y sin limitarle la profundidad a cada arbol - tendríamos que hacer pruebas variando esos parametros. Acá nos enfrentamos con otro problema - la implemetación de *scikit* de RandomForest consume demasiada memoria RAM - dicho entrenamiento y posterior predicción usaron los 6GB de una notebook. Por otro lado, la aplicación fue muy rápida, tardando unos pocos minutos.

4.3.2. Support Vector Machine

El método conocido bajo este nombre es un modelo para aprendizaje supervisado. Este modelo tiene algoritmos asociados, que reconocen analizan los datos y reconocen patrones, mediante los cuales SVM intenta representar los datos de entrada cono puntos en el espacio, para que los datos de cada categoría queden claramente separados.

Hicimos varias pruebas sobre este algoritmo, pero no tuvimos mucho exito - la implementación de scikit (y SVM en general) no está preparada para clasificar nuestra cantidad de datos. Con tan solo un subset de 40000 ejemplos el algoritmo tardaba en aprender varios minutos.

4.3.3. Predicción sin algoritmo

La predicción sin algoritmo tambien es un enfoque posible - y es bastante sencillo, es el camino que tomamos al principio para entender mejor de que se trata la competencia. La idea es realizar una predicción basada en estadísticas y promedios del set de entrenamiento. Por ejemplo, asumir que la probabilidad de cada crimen de una fila del set de prueba es igual al procentaje de cada crímen en el distrito de esa fila. Ese porcentaje es calculado anteriormente sobre el set de entrenamiento - basicamente es el total de tipo de crímen sobre el total de crimenes en el distrito. Es interesante que este método, solo con el distrito, recibió un *logloss* de 2.65400 en Kaggle.

Tal resultado obviamente se debe a la distribución variada de crímenes donde unos pocos tipos de crímen conforman el mayor porcentaje, y la puntuación va a ser alta mientras esos crímenes tengan mayor probabilidad. Sin embargo, seguramente el resultado va a mejorar agregando más estadísticas. Este enfoque tiene sus problemáticas, como el descarte de los falsos negativos, pero es rápido y fácil. Se puede tomar como piso y tratar de mejorar sus resultados aplicando las prácticas de Machine Learning.

5. Implementación en C++

Para que la implementación de los algoritmos sea eficiente, la mayor parte de los resultados a utilizar fueron expresados en forma matricial, de forma de poder aprovechar las alta performance en estas operaciones de bibliotecas como *Armadillo*. Esta última, que es la que estamos usando en este momento, tiene además un tipo de datos SpMat especialmente destinado al trabajo con matrices dispersas, lo cual puede ser de particular utilidad por el acercamiento que tomamos respecto a las features discretas.

Desde el punto de vista de diseño, tendriamos que dividir la solución en pasos del flujo de datos, desde la entrada del archivo csv inicial, hasta el archivo csv con las probabilidades calculadas, pasando por la limpieza y conversión de datos, aprendizaje y predicción de los algoritmos, posible promedio entre las salidas de los algoritmos. La ídea es hacer que esos pasos sean separados e independientes, facilitandonos de esta manera las mediciones de tiempos y búsqueda de errores. Idealmente la salida de cada paso tendría que poder ser guardada, para luego seguir desde ese punto. Probablemente necesitemos alguna libreria que pueda parsear eficientemente un csv de entrada y serializar los datos en forma de csv tambien.

Referencias

[1] Foros de la competencia de Kaggle
https://www.kaggle.com/c/sf-crime/forums

[2] Neural Networks and Deep Learning, de Michael Nielsen, Agosto del 2005. Capítulo 2 - How the backpropagation algorithm works.

[3] Andrew NG, Co-fundador de Coursera, Prof. Universidad de Stanford. *Video-tutoriales de curso de Machine Learning de Coursera.*

[4] Breve explicación de las herramientas matemáticas y algorítmicas utilizadas por los algoritmos de sklearn (python)
http://scikit-learn.org/stable/modules/

[5] Wikipedia [en] *Imagen ilustrativa - Coordinate Descent*

[6] Video de Andrew Ng en Coursera sobre el descenso por el gradiente estocástico. https://youtu.be/HLf4QNAwsd0