**VISUALIZACIONES**

Scatter plot: Nos sirven para darnos una idea de la dependencia que existe entre las dos variables y de las características de esta dependencia. El color puede representar una variable tanto categórica como numérica en cuyo caso será un gradiente de color. Los ejes x e y deben ser variables numéricas

Bar plot: Una de las variables tiene que ser categórica, y los ejes deben comenzar en 0. La variable categórica es aquella por la cual vamos a crear las barras.

Histograma y Density plot: Sirve para mostrar la distribución de una determinada variable, es decir la cantidad de veces que la variable toma determinados valores. Se necesitan dos cosas, la variable en cuestión que tiene que ser numérica (continua o discreta) y el ancho que van a tener las columnas del histograma. La variable x es numérica, y siempre esta ordenada, el eje y siempre es una cantidad.

Box plot: Una forma de visualizar la distribución de una variable numérica, se usa para comparar la distribución de la variable de acuerdo a una variable categórica. Poner muchos box plot uno al lado del otro es un recurso muy útil para entender si una variable afecta la distribución de otra.

Violin plot: Es la solución cuando en un boxplot no se puede permitir comparar claramente cual de las variables tiene mas o menos outliers. Nos muestra un ancho correspondiente a la densidad de la variable para el valor del eje y correspondiente.

Grafico Lineas: Son muy útiles pero tiene la limitación de que en casi todos los casos el eje x debe ser tiempo. La variable en el eje y puede ser de cualquier tipo aunque siempre es ordinal, ya sea numérica (el caso más frecuente) o categórica. El eje x representa la evolución del valor de dicha variable a lo largo del tiempo.

Heatmaps: Es una matriz en donde cada celda muestra el valor que toma la variable del eje x para el punto del eje y.

**PANDAS**

Map: Para aplicar una función a cada elemento de una columna – col.map(f) -> col

Apply: Aplicar una función a la columna entera, la función procesa la columna entera y puede devolver o bien un dato atómico o bien una nueva columna – col.apply(f) -> any

**SPARK**

Transformaciones: Transforman un RDD en otro. Son lazy, es decir, que solo se ejecutan cuando se realiza una acción. Se pueden cachear, es decir guardar en el cache esa transformación para evitar recalcularla.

* Map
* Filter
* FlatMap
* reduceByKey
* GroupByKey
* Join

Acciones: Devuelven un valor, las transformaciones se pueden encadenar, las acciones no.

* Reduce
* Collect
* Count
* CountByKey
* Take
* TakeOrdered
* First

**NLP**

El problema de representación intenta resolver la duda de como representar de forma tabular algo que no lo es, pasa con cualquier cosa que nosotros necesitemos representar.

Primero tenemos como representar texto, con los modelos de Bag of Words y el modelo TF-IDF

En bag of words tenemos una columna por cada una de las palabras que nosotros decidamos que puede ser interesante, y pondríamos un uno y un cero para indicar si esa palabra se encuentra o no.

En TF-IDF el método es similar, lo que cambia es que el numero que va por cada palabra surge de hacer el producto entre el peso del termino en el documento (puede ser la frecuencia de ese termino) y el idf, que es una noción de la relevancia que tiene el termino (que tan raro o importante es de acuerdo a los documentos). Se puede decir que tf-idf le asigna menor peso a las palabras mas probables.

Si no quiero seleccionar las palabras, quiero seleccionar la cantidad de columnas que voy a tener lo hago usando una función de hashing, esta función me dice donde colocarla y ahí pondré el valor ya sea de bag of words o tf-idf. Esto puede producir colisiones por lo que a veces se usa una segunda función de hashing. Tambien puedo usar embeddings.

Para procesar los textos hay varias cosas para hacer, como decidir que hacer con los stopwords, palabras muy comunes que a veces quiero eliminar, y si lematizar, que es convertir algunas palabras a su raíz de acuerdo a su idioma (ponerlas en infinitivo).

A medida que resumimos el contexto de las palabras podemos encontrar su significado, esa es la teoría lenguistica en el nlp moderno. Para hacer esto se construyen Word embeddings.

Cuando se crea el embedding se va a tener un vector por cada palabra, entonces yo le paso una serie de palabras y mapea cada una de esas palabras a su vector. El embedding no es mas que una matriz o un diccionario, le pasas una palabra y la transforma en un vector.

**MACHINE LEARNING**

Hay dos tipos, los problemas de regresión, es decir, donde la variable a predecir es continua (un numero que puede tomar cualquier valor), y los problemas de clasificación, en el que lo que queremos aprender tiene un valor categórico (una serie valores posibles)

Ej regresión: Precio de una propiedad – Ej Clasificacion: Clasificar una imagen

El set de datos tiene filas (rows u observaciones) y columnas (features o variables), la columna que queremos predecir la llamamos label o target.

Al proceso de creación de features lo llamamos “feature engineering”, en el que definimos la representación que le vamos a dar a cada dato.

El objetivo es entrenar un modelo de Machine Learning que nos permita predecir el label a partir de los feature.

Test tiene que tener distribución productiva de los datos, train no.

La validación del modelo se hace dividiendo el set de datos en un set de test y uno de entrenamiento, el de entrenamiento lo usamos para entrenar y el set de test lo usamos para medir la performance de nuestro modelo.

El entrenamiento de un modelo es un problema de optimización, buscamos los parámetros óptimos para minimizar o maximizar la métrica elegida. Dependiendo del modelo varia la técnica de optimización a usar.

Los hyperparametros son valores que parametrizan el entrenamiento del modelo, son valores que debemos indicar por afuera, no son parte del proceso de optimización del modelo. A la búsqueda de hyperparametros se lo conoce como tunear un modelo.

Para la búsqueda de hyperparametros tenemos varias técnicas que nos solucionan esto. Por ejemplo Gird-Search que es búsqueda por fuerza bruta, establecemos un conjunto de valores a probar por cada hyperparametro, prueba cada combinación y se queda con la mejor, si tengo por ejemplo dos grillas de parámetros que pueden tomar tres valores cada uno, se probaran en total nueve casos. Otro es Random Search, la alternativa rápida, funciona igual que Grid Search pero en vez de buscar todas las combinaciones posibles busca una cantidad fija de combinaciones al azar, probamos “k” combinaciones aleatorias de hyperparametros y nos quedamos la mejor, decidimos cuanto tiempo invertir pero no probamos todas las combinaciones. En la búsqueda bayesiana, es como un Random Search pero en vez de ir probando todas de forma aleatoria, lo hace de acuerdo a como le fue en pruebas anteriores, es la mejor en cuanto a velocidad-resultados.

Para probar los hyperparametros se divide el set de entrenamiento en dos: entrenamiento y validación. Set de entrenamiento es solo para entrenar. Set de validación es para encontrar los mejores hyperparametros para mi algoritmo. Set de test es para ver como le va. Para solucionar fallos los modelos (esto es algo que hacen internamente, no nosotros) de machine learning hace cross-validation en el que dividen el set de entrenamiento en “k” particiones del mismo tamaño, y usan cada una de esas particiones como set de validación entrenando con el resto y promedia la métrica en las “k” validaciones.

Underfitting se produce cuando tenemos un error de entrenamiento (es decir, midiendo el score con el set de entrenamiento) alto. Eso quiere decir que el modelo se ajusta mal al set de entrenamiento, el modelo no tiene suficiente expresividad para los datos. La única solución es cambiar el modelo a uno mas complejo y expresivo.

Overfitting se produce cuando tenemos muy buen resultado para el set de entrenamiento pero no tan bueno para el set de test. Esto pasa porque el modelo es demasiado expresivo, tiene demasiados grados de libertad y se termina aprendiendo de memoria el set de entrenamiento, por lo que se dice que el modelo generaliza mal, el modelo es más expresivo. La solución es disminuir la complejidad del complejo o conseguir más datos.

KNN es básicamente un modelo que aprende por analogía. Puede usarse tanto para clasificar como para regresiones. Lo que hace es revisar los k vecinos mas cercanos que tiene, y respecto a eso decide de que elemento se trata, por ejemplo, si la incógnita tiene dos círculos cercanos y un cuadrado, y tomamos un k igual a 3, la incógnita será un circulo ya que tiene próximos dos círculos y un cuadrado. El tiempo de entrenamiento es nulo, ya que no necesita entrenarse, por esto puede decirse que es el algoritmo mas rápido. En la clasificación hay un problema de performance, ya que para buscar los puntos mas cercanos a cierto punto implica buscar linealmente entre todos los datos. La pregunta es como encontrar el mejor valor de k (cuantos vecinos tomar), cuando k es pequeño, el algoritmo tiende a hacer overfitting, si k es muy grande, hace underfitting. Para encontrar el k simplemente hay que ir probando, usando cross-validation por ejemplo. También es necesario determinar la distancia a usar (que no siempre es la distancia euclidiana). Los atributos tienen que estar normalizados (escalar cada columna al rango 0/1) de lo contrario un atributo puede dominar todas las distancias. También es sensibles a features ruidosos, por ejemplo, predecir la altura de una persona. Una practica normal es reducir el peso del atributo o aprender los pesos óptimos.

La maldición de la dimensionalidad es la que dice que no todos los algoritmos funcionan bien en cualquier cantidad de dimensiones. Si la cantidad de dimensiones es muy grande entonces la distancia entre dos puntos cualesquiera converge. (KNN tiene problemas de performance, no la maldición). Cuanto mas van aumentando las dimensiones la variabilidad va desapareciendo, esto hace que los algoritmos de machine learning basados en distancias tengan muy difícil el saber que puntos están mas cerca que otros.

Para las redes neuronales se usan varias capas, la ventaja de esto es la abstracción, cada capa se abstrae mas de los datos iniciales, obteniendo mejores representaciones de los datos. La idea es usar muchas capas, lo que es Deep learning, la dificultad de esto es el problema del gradiente, pero si se las entrena bien pueden llegar a tener muchos mejores resultados.

Un árbol de decisión es un árbol en el que cada nodo se hace una pregunta. Funcionan tanto con datos numéricos o categóricos, requieren poca preparación de los datos (no requieren normalización), tienen buena performance para datasets grandes y ayudan en la selección de features. Las desventajas es que se usan algoritmos greedy para encontrar el árbol optimo, y otra es que los arboles demasiados complejos generan overfitting. En arboles de decisión, todos los datos que incluyan interacción entre features son importantes. Ya que estos modelos ven una columna a la vez, no pueden ver dos columnas al mismo tiempo.

Algunos tipos de árboles de decisión son:

* ID3: Algoritmo de tipo gready, en cada paso intenta realizar el mejor Split en base al atributo que nos da mayor ganancia de información (relación con la entropía). Esto se repite recursivamente. Features categóricos. También tiene gran peligro de overtiffing.
* C4.5: Evolucion del ID3, acepta atributos numéricos y acepta datos con atributos faltantes, los atributos pueden tener un peso. Poda el árbol generando un árbol mas simple
* CART: Usados para clasificación o regresión, utiliza la medida Gini de impureza para determinar el Split, este Gini es una medida en base a la probabilidad de clasificar mal un dato utilizando la distribución de probabilidades del target.

Los ensambles son los mejores algoritmos de machine learning, básicamente es usar combinaciones de varios algoritmos, aunque no siempre sea lo mejor para usar en la práctica. Algunas de las técnicas para realizar el ensamble es el bagging y el boosting.

El bagging es aplicar el mismo clasificador n veces distintas y luego obtener un promedio de sus resultados, esto lo hace usando Bootstrapping, que es tomar muestras del set de entrenamiento de igual tamaño que este. El bagging disminuye la posibilidad de overfitting, esto es porque cada clasificador no ve la totalidad de los registros del set de entrenamiento. Y también se pueden utilizar los registros Out of Bag, usando los datos que no utilice para ver la precisión del algoritmo.

El boosting es entrenar un algoritmo simple, analiza esos resultados y entrena otro algoritmo simple en donde empieza a mejorar un poco mas lo que tenia antes, y asi se van encadenando distintos algoritmos en los que voy mejorando en cada uno los resultados para así al aplicar varios, obtener un buen resultado. Es decir, que genera un árbol que busca predecir bien lo que predijo mal el árbol anterior. La profundidad de los arboles generados con boosting es menor a la generada con bagging.

Random Forest es uno de los algoritmos mas populares en clasificación. En éste se trabaja con bagging, generando varios arboles de decisiones, en el que cada uno se construye generando un Bootstrap del set de entrenamiento y un subconjunto de los atributos. Tienen bastantes hyperparametros, uno es la cantidad de arboles a crear, mas arboles dan mejores resultados pero también es mayor utilización de recursos, también la cantidad de atributos por árbol.

XGBoost un boosting de arboles de decisión, genera arboles simples. Para atributos numéricos se ordenan los valores del atributo y se busca el Split de mayor ganancia. Los atributos categóricos deben ser convertidos a variables binarias usando algún encoding ya que solo funcionan con valores numéricos. Hay otros algoritmos similares que son mejoras de este que son por ejemplo CatBoost (los arboles tienen que estar balanceados) y LightGMB, tienen optimizaciones de uso de memoria, velocidad y entrenamiento en paralelo, y soportan variables categóricas mediante un encoding interno.

**REDUCCION DE DIMENSIONES**

Se busca combatir la maldición de la dimensionalidad, para poder estudiar si un algoritmo mejora o empeora cambiando la dimensión de los datos. También es para la performance, ya que podemos usar un algoritmo inviable con nuestro set de datos por la cantidad de dimensiones que tiene, al mismo tiempo adaptarlo por si tengo recursos limitados, facilitando el almacenamiento y el procesamiento de datos.

También se puede usar como un mecanismo de feature engineering. Como reducir el nuevo set de datos filtrando features ruidosos y devolviendo un set con una menor cantidad de ruido, esos features pueden agregarse a los otros, aumentando el nivel de señal de los datos, y hasta descubrir correlaciones o tópicos ocultos en los datos.

También para la interpretación y visualización de datos. Ya que el cerebro humano esta entrenado para entender datos de dos o tres dimensiones.

SVD: Toda matriz se puede descomponer como el producto de tres matrices, sabemos que los vectores de V están ordenados según su importancia. Podemos tomar los primeros k vectores para representar los datos. La SVD nos da la mejor aproximación de rango k posible a la matriz original.

Dimensión Intrínseca de los datos: La clave es que épsilon (suma) contiene los valores singulares ordenados de mayor a menor. Podemos calcular la energía que conservamos usando los k valores singulares mas significativos.

PCA: Es un método lineal, en este método queremos encontrar componentes (un vector) que maximicen la varianza de los datos. Puede ser vista como una rotación del espacio, para entrar un eje que mejor exprese la variabilidad de los datos. El método consiste en normalizar los datos para centrarlos, calcular la matriz de covarianza, y calcular autovalores y autovectores de la matriz de covarianza. PCA = SVD (normalizada).

T-SNE: Partimos calculando la probabilidad de que un punto sea vecino de otro. Buscaremos que, si dos puntos son cercanos en el espacio original, lo sean también en el reducido.

**CLUSTERING**

Tiene muchos usos además del agrupamiento de datos, como para entenderlos, mejorar los resultados de un modelo o mejorar en tiempo algún problema que tengamos.

Los algoritmos de clustering agrupan automáticamente (no supervisado) grupos de elementos en “clusters”. El objetivo es que cada clúster sea internamente coherente pero diferente a los demás clusters, es decir, que los elementos pertenecientes a un mismo clúster deben ser muy similares y al mismo tiempo muy distintos a los elementos en los demás clusters.

Clustering jerarquico: Suele generar soluciones muy buenas y no escala. Partimos de que cada punto es un clúster, y en cada paso unimos los dos clusters mas cercanos, repetimos este proceso hasta que tengamos un único clusters o la cantidad que queríamos.

k-means: dado un set de puntos en d dimensiones, particionar los puntos en k sets siendo k menor a d, de tal forma de minimizar la sumatoria de las distancias de cada punto al centro del clúster. Elegimos k puntos, uno para cada cluster, inicialmente los centros de cada cluster son esos puntos, luego le asignamos a cada punto el cluster al centro mas cercanos, debo fijarme cual es el centro mas cercano al punto que tengo y a ese le asigno el cluster, el problema de hacer esto es que hay que volver a re calcular los centros como el promedio de los puntos, asi que volvemos al paso anterior y repetimos la cantidad de iteraciones que elegimos. Para encontrar k se puede hacer por ejemplo el promedio de la sumatoria de distancias dentro de cada cluster. K-means es bastante sensible a los puntos iniciales, si tenemos una mejor forma de encontrar los puntos iniciales la solución optima se encontrara mucho mas deprisa.

Clustering espectral: Es aplicar k-means sobre Laplacian Eigenmaps, esto es hacer una matriz de afinidad, a esto el laplaciano, calcular la descomposición SVD y después quedarnos con los m autovectores mas pequeños haciendo la reducción de dimensiones.