

Comparación de rendimiento de predictores

de meses según la climatología de la Comunidad de Madrid

Sistemas inteligentes

Félix García Narocki, David Ferreras Díaz, Pablo Padial Iniesta

[**Resumen 4**](#_3v2205ookef6)

[Objetivo 4](#_wf0rzqyqo0df)

[**Introducción 4**](#_9t8pa8wq9l3p)

[El dataset y limpiar datos 4](#_43k6udvi5mn9)

[**Algoritmos entrenados 5**](#_xwsc9fgmtbii)

[KNN (K-Nearest Neighbors) 5](#_9yy5j53ut9rh)

[Descripción 5](#_ab22qtwh1ydx)

[Implementación 5](#_zacgixas0oku)

[Matriz de confusión 6](#_ifrela1ohidc)

[Precisión 7](#_q1sk055typnz)

[Logistic Regression 7](#_i331ei12m2ib)

[Descripción 7](#_5qrohc661i1m)

[Implementación 7](#_y169icharvpo)

[Matriz de confusión 8](#_z3mntlwabqh4)

[Precisión 8](#_bu2ykmfskux3)

[Support Vector Classifier (SVC) 9](#_d2pp78krd8ig)

[Descripción 9](#_h2ravjnao4c2)

[Implementación 9](#_x4liq7didp89)

[Matriz de confusión 10](#_dvu1gp1eyo9f)

[Precisión 10](#_khsmt7tud3gw)

[Decision Tree 11](#_1uq7axw1gp4p)

[Descripción 11](#_do2ybjebx424)

[Implementación 11](#_aejt3b4qtyoq)

[Matriz de confusión 11](#_r4eykzpsgab1)

[Precisión 12](#_qvds22ehvocz)

[Random Forest 12](#_hnqgehdcu3bn)

[Descripción 12](#_f93f9tn7qlbk)

[Implementación 12](#_6gdl5vds8z1y)

[Matriz de confusión 13](#_22l1xu4oedtc)

[Precisión 13](#_bynxiocfygi)

[GridSearch + RandomForest 14](#_c1ay6cmppcwc)

[Descripción 14](#_x8e14mzawzhb)

[Implementación 14](#_sd3i8yjeab3t)

[Matriz de confusión 14](#_f24055781szs)

[Precisión 14](#_92u5f8fqaegr)

[Gaussian Naive Bayes (GaussianNB) 15](#_mp70syeiro5k)

[Descripción 15](#_166czkyopwlh)

[Implementación 15](#_9qwnyhsfi8ae)

[Matriz de confusión 15](#_tnhmz6c58903)

[Precisión 16](#_n1tjtdd73xe0)

[Algoritmo genético + Perceptron 16](#_60xy1030mazr)

[Descripción 16](#_dx0cqwfkzj5v)

[Implementación 16](#_55s4vbq5xmbx)

[Preparación 16](#_efhq4as3vzlw)

[Creando el algoritmo 17](#_u3kvejauvrs2)

[Ejecución del algoritmo 18](#_lw48wns0ptoi)

[Entrenando el modelo final 19](#_d5th1mgl3058)

[Precisión 19](#_wn275ggrjwmf)

[Matriz de confusión 19](#_4bphfndjv3jm)

[**Resultados 20**](#_lzk17f7f5p0m)

[Bibliografía 22](#_20ejkybaemjt)

# 

# Resumen

Este es un proyecto que pretende encontrar el modelo de aprendizaje automático que pueda predecir mejor el mes del año cuando le damos como entrada las variables climatológicas de la Comunidad de Madrid

# Objetivo

El objetivo de esta práctica es encontrar un dataset, limpiarlo y probar varios algoritmos, entre ellos un perceptrón, para poder comparar resultados. En nuestro caso hemos elegido un dataset de los datos climatológicos de la comunidad de Madrid y entrenar una red neuronal para que prediga el mes en base a los datos meteorológicos que le introducimos.

Para los algoritmos decidimos crear primero un algoritmo genético para probar distintas arquitecturas del perceptrón antes de hacer el entrenamiento final, con el objetivo de encontrar el número de capas, las neuronas, y la función de activación entre capas que mejor resuelva el objetivo.

# Introducción

La dificultad sobre esta práctica ha sido el tipo de datos con el trabajamos. Los datos climatológicos son muy variables pues el comportamiento climático no sigue un patrón fijo mes a mes. Existen grandes diferencias entre estaciones, pero no tanto entre meses adyacentes y además, por la existencia de una simetría entre otoño y primavera, descubrimos que es donde el modelo presenta algunos problemas para distinguir entre unos meses y otros.

Para mostrar los resultados de cada modelo individualmente, mostraremos su matriz de confusión, que nos contrasta el número de aciertos en la predicción con el mes real con el que trabajamos. Aunque debido a la variabilidad del clima mencionada, hemos decidido trabajar no solo con la opción de más confianza del modelo, si no también con el top 3. Esto nos dará una mejor imagen para evaluar el rendimiento pues queremos saber si igual se confundió con un mes adyacente o uno “simétrico” de otra estación.

# El dataset y limpiar datos

Como mencionamos anteriormente, el dataset está compuesto por los [datos climatológicos de la comunidad de Madrid](https://www.madrid.org/iestadis/fijas/coyuntu/otros/clpreci.htm) desde 1895 hasta la actualidad. La comunidad de madrid nos proporciona tres tipos de dataset, precipitaciones, temperatura y presión. De cada uno de ellos los separa en datos históricos [1895-1985] y actuales [1985-2012].

Lo primero fue juntar los dataset, primero juntamos los tres dataset con datos históricos y a continuación los tres con datos actuales, para al final, juntarlos para tener un único dataset con todos los datos climatológicos de 1895 a 2012.

Los datos que nos proporcionan actualmente tienen más atributos que los históricos, con muchos datos faltantes. Además, analizando alguno de estos atributos, decidimos cuales son los atributos más relevantes para nuestro objetivo, por ejemplo descartamos las columnas como “Hora de medición” ó “Día de mayor precipitación”, en este caso está claro que no son un atributos que ayuden a determinar en qué mes estamos.

Dentro del dataset, encontramos que en algunas columnas, algunos valores no eran numéricos, por ejemplo en las precipitaciones el valor “Ip” (Inapreciable) hemos tenido que transformarlo en 0.0.

Debido a la cantidad de datos nulos, hemos tenido que hacer una ponderación para eliminar filas y columnas en las que faltan demasiados datos. Para empezar, sabiendo que contamos con alrededor de 1300 filas, eliminamos las columnas que tuviesen más de 800 filas nulas, pues complicaría los siguientes pasos. Lo siguiente fue eliminar las filas que tengan más de un 50% de nulos. Con las columnas y filas restantes, decidimos rellenar los datos que faltan usando un algoritmo con el que usando la mediana, la desviación típica y una función de ruido, reemplazar los nulos por el valor computado.

Así, para un dataset que tenía 35 columnas y 1341 filas, nos quedamos con un dataset limpio con 17 columnas y 1095 filas.

# Algoritmos entrenados

## **KNN (K-Nearest Neighbors)**

### Descripción

Este algoritmo no tiene un entrenamiento al uso como el resto que veremos. KNN, como su nombre indica lo que hace es aproximar la respuesta según cuales son los k datos más cercanos. Dicho de otra forma, al entrenar guarda los datos y los coloca en el espacio. Cuando un nuevo dato llega, mide las distancias desde el nuevo a los k más cercanos usando un algoritmo para medir la distancia, que puede ser Manhattan, Euclídea, etc…

Entonces se predice que el nuevo dato será parecido al de sus vecinos.

### Implementación

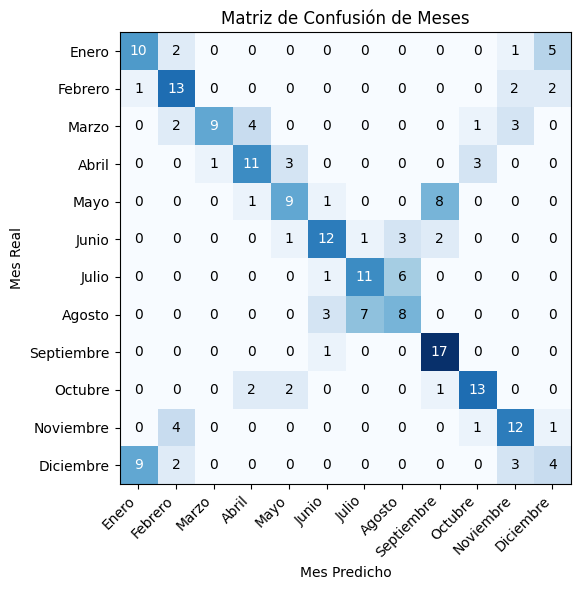
Para todos los siguientes algoritmos hemos usado la misma separación de datos, sin usar LabelEncoder ni OneHotEncoder ya que no hacen falta para los algoritmos de SkLearn de aprendizaje automático.

Para el modelo KNN hemos normalizado el entrenamiento y usado distancias Manhattan con votos ponderados y paralelismo. Sus parámetros han sido los siguientes:

Hemos usado 15 vecinos con el algoritmo de árbol de knn. Lo hemos configurado con un tamaño de hojas de 20.

Tras entrenar el modelo con fit hemos sacado su tiempo de ejecución de milisegundos, su precisión de 0.5890 global y su top 3 aciertos. Es el algoritmo más veloz después del Gaussiano a cambio de ser muy impreciso comparado con el resto, quedando solo por delante de nuestro siguiente algoritmo, Logistic Regresion.

### Matriz de confusión



### 

### 

### Precisión

|  | Top 1 | Top 2 | Top 3 |
| --- | --- | --- | --- |
| Precisión | 59,36% | 84,47% | 94,98% |

## **Logistic Regression**

### Descripción

Este algoritmo trata de usar el descenso del gradiente para encontrar la solución óptima. Se diferencia de una regresión lineal en la ecuación logística, en este caso es de la forma:

### Implementación

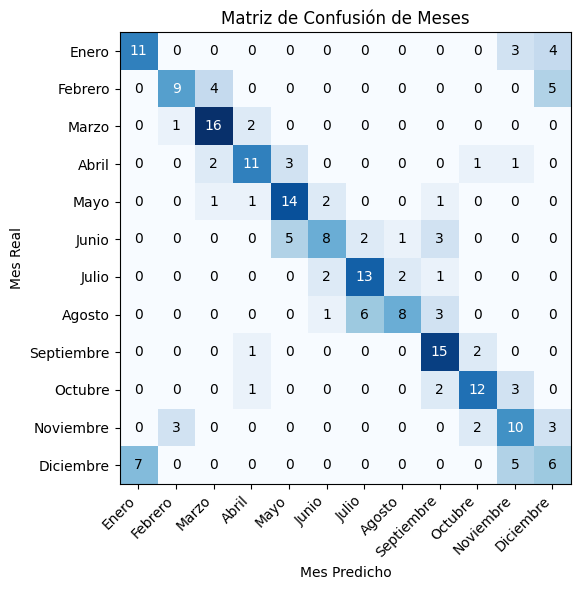
Para el algoritmo Logistic Regresion lo primero que hemos hecho ha sido realizar una estandarización, con StandardScaler(), para que las características tengan una media cero y desviación estándar unitaria. Utilizamos los parámetros:

Utilizamos un clasificador de regresión “ovr” con solver “lbfgs” un máximo de 2000 iteraciones con todos los núcleos disponibles.

Tras entrenar el modelo sacamos su tiempo de ejecución en milisegundos, su precisión de de 0.6073 junto con su top-3 y su matriz de confusión. Este algoritmo es el menos preciso de todos aunque sigue siendo de los más veloces.Matriz de confusión

### Matriz de confusión

### 



### Precisión

|  | Top 1 | Top 2 | Top 3 |
| --- | --- | --- | --- |
| Precisión | 58,90% | 84,93% | 92,69% |

## 

## **Support Vector Classifier (SVC)**

### Descripción

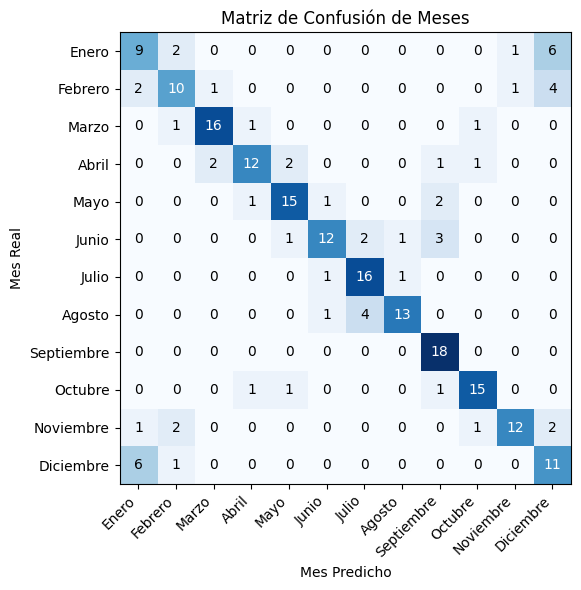
Este algoritmo trata de formar la frontera de decisión para la predicción, añadiendo una nueva dimensión, usando esta, crea un hiperplano que permite separar los datos en sus categorías.

### Implementación

El siguiente algoritmo es Support Vector Machine donde también realizamos una estandarización al igual que en Logistic Regresion. Para este algoritmo usamos los siguientes parámetros:

Para el kernel utilizamos “rbf” el utilizado por defecto que usa la distancia euclídea. También el valor de C, que mide la penalización, lo estuvimos ajustando hasta encontrar que un valor 10 era lo más óptimo. Activamos el cálculo de probabilidades y utilizamos el valor por defecto de gamma “scale”.

Este es un algoritmo que nos sorprendió mucho debido a su relación tiempo-precisión, consiguiendo una de las mayores precisiones en poco tiempo.

Matriz de confusión

### Precisión

|  | Top 1 | Top 2 | Top 3 |
| --- | --- | --- | --- |
| Precisión | 73,52% | 89,94% | 98,17% |

## **Decision Tree**

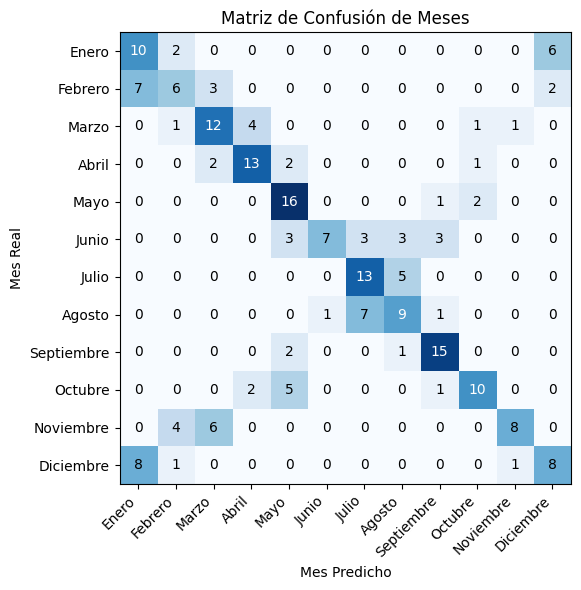
### Descripción

Los árboles de decisión funcionan con “preguntas” en cada nodo del árbol. Se toma una rama dependiendo del valor que se esté valorando.

### Implementación

Para este algoritmo, no es necesario normalizar ni aplicar OneHotEncoder ya que al normalizar datos el algoritmo cambia el umbral y por lo tanto el árbol construirá exactamente las mismas particiones. Si aplicas OneHotEncoder, este puede generar overfitting ya que el árbol hará comparaciones de igualdad o rango.

Para configurarlo hemos usado una profundidad máxima de 5 acompañado con una división mínima de muestras de 4.

Matriz de confusión

### Precisión

|  | Top 1 | Top 2 | Top 3 |
| --- | --- | --- | --- |
| Precisión | 58,90% | 79,00% | 88,13% |

## **Random Forest**

### Descripción

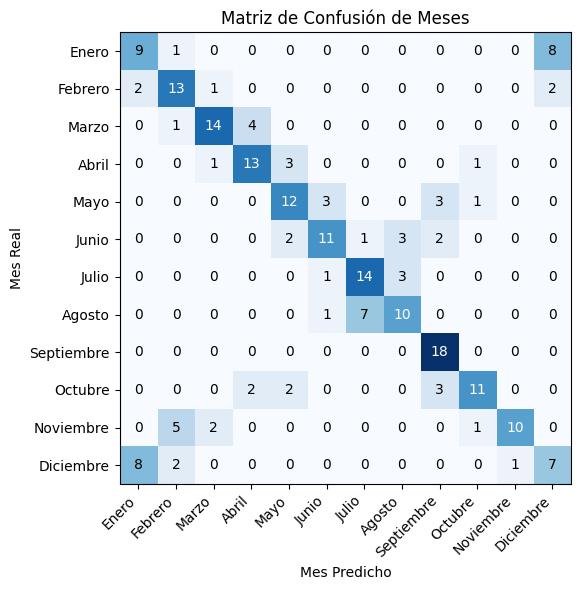
Esta es una extensión de los árboles de decisión, pues emplea varios árboles para seleccionar la predicción en base a la media de todos los árboles.

### Implementación

Ahora nos toca el algoritmo Random Forest, en fuimos modificando la cantidad de árboles hasta dejarlo en 100 donde comprobamos que tenía el mejor rendimiento y precisión. Del mismo usamos una profundidad máxima de 15. El mínimo número de muestras que debe tener un nodo interno lo limitamos a 2 mientras que el número mínimo de muestras de cada lo dejamos en 1, además, utilizamos todos los núcleos que nos permite la máquina para así acelerar el proceso.

Este algoritmo consigue una muy precisión, sobre todo cuando se mide el top-3, siendo el segundo algoritmo más preciso y aunque tarde algo más que los anteriores aun sigue muy lejos de los genéticos. Esto hace que pueda ser uno de los mejores algoritmos en cuanto a la relación tiempo-precisión.

### Matriz de confusión



### Precisión

|  | Top 1 | Top 2 | Top 3 |
| --- | --- | --- | --- |
| Precisión | 63,93% | 89,04% | 98,17% |

## **GridSearch + RandomForest**

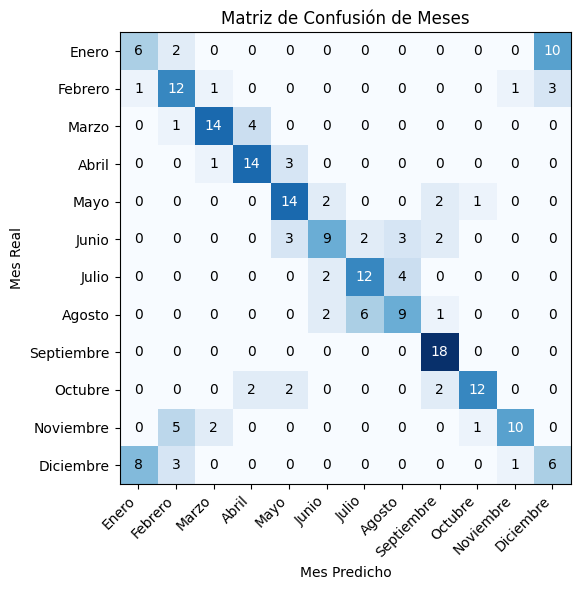
### Descripción

El grid search junto con el Random Forest, trata de buscar los hiper parámetros del árbol más óptimos dado el problema, probando cada Random Forest con combinatoria.

### Implementación

Para usar GridSearch necesitamos hacer LabelEncoder en el y\_train e y\_test. Definimos los parámetros que queremos probar en el param\_grid junto a un TimeSeriesSplit con valor 5 en lugar de un K-Fold para que cada partición respete el orden cronológico y evite mirar al futuro. Para el scoring hemos usado MSE convertido a negativo ya que GridSearch los maximiza. Al final imprimimos los tres mejores resultados con sus parámetros implementados en su ejecución.

### Matriz de confusión



### Precisión

|  | Top 1 | Top 2 | Top 3 |
| --- | --- | --- | --- |
| Precisión | 62,10% | 87,21% | 98,17% |

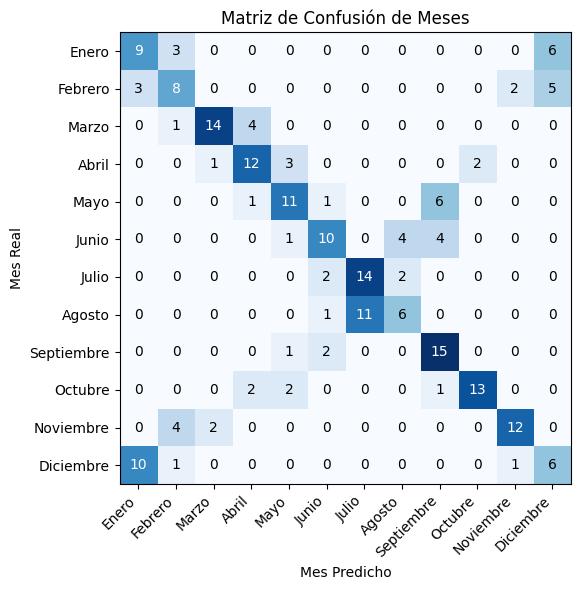
## **Gaussian Naive Bayes (GaussianNB)**

### Descripción

El modelo Gaussian Naive Bayes está diseñado para trabajar con variables numéricas continuas, asumiendo que cada característica sigue una distribución normal gaussiana. Para ello usa un clasificador probabilístico que aplica el teorema de Bayes. Este modelo es muy rápido de entrenar y presenta una predicción instantánea.

### Implementación

Este algoritmo necesita datos de entrenamiento normalizados para funcionar con precisión pero no depende del OneHotEncoder. Se caracteriza por ser muy simple y fácil de implementar ya que no necesita ningún parámetro, solo una buena limpieza de los datos de entrada y un entrenamiento correcto.implementar ya que no necesita ningún parámetro, solo una buena limpieza de los datos de entrada y un entrenamiento correcto.

Matriz de confusión

### Precisión

|  | Top 1 | Top 2 | Top 3 |
| --- | --- | --- | --- |
| Precisión | 59.36% | 83,11% | 97,26% |

## **Algoritmo genético + Perceptron**

### Descripción

Como hemos trabajado con los dos algoritmos, explicaremos los dos aquí.

El algoritmo genético es un algoritmo de búsqueda, se usa para encontrar soluciones a problemas usando selección natural. Se crea una población de individuos que a lo largo de generaciones se cruzan y mutan, seleccionando a los mejores en cada una para que ‘pasen’ sus genes a la siguiente generación de individuos.

Cada individuo se evalúa para darle un fitness, que es una medida para encontrar cuánto se acerca a la solución.

Un perceptrón es un algoritmo basado en redes neuronales, que intenta imitar el comportamiento del cerebro. Dada una entrada y una salida esperada se crean capas de neuronas. Estas neuronas están conectadas de una capa a otra todas con todas, dependiendo de la entrada, unas neuronas serán activadas y otras no, activando al final las neuronas que serán las salidas.

La red se entrena introduciendo datos (feedforward) y con el output recibido se ajustan en función de si acierta o no (feedbackwards). Después de varias iteraciones por los datos se obtiene una red neuronal ajustada al problema.

### Implementación

Con esta combinación tratamos de hacer una búsqueda de la mejor arquitectura para una red neuronal, y para ello hemos usado un algoritmo genético para que pruebe y seleccione una configuración lo más óptima posible.

#### Preparación

Para esta fase hemos usado SKLearn para la preparación de los datos y Keras de Tensor Flow para construir y entrenar los modelos.

Lo primero para diseñar un algoritmo genético es definir los genes. En nuestro caso, un gen está formado por el número de neuronas y una función de activación entre relu, tanh y sigmoid. La longitud del genotipo (Conjunto de todos sus genes) determina el número de capas de la red.

Además el objetivo es poder entrenar un modelo que prediga el mes en el que estamos, por lo que hemos separado el input y el output por la columna ‘Mes’, siendo X el resto de columnas y Y el mes.

Para los meses, son datos categóricos, por lo que no podemos utilizarlos directamente, si no que antes tenemos que codificarlos en One-Hot encoding para que a cada mes le corresponda un número sin asignarle un orden, para no introducir sesgos al modelo.

Después hemos separado los datos en entrenamiento y test (80, 20) y una segunda división de los de entrenamiento para evaluar los individuos entre entrenamiento y validación (85, 15).

#### Creando el algoritmo

Ahora que tenemos la genética definida y los datos separados, el siguiente paso es decidir cómo se crean los primeros individuos, cómo seleccionamos entre generaciones a los mejores, como se mezclan los genes entre generaciones, cómo ocurren las mutaciones y cómo evaluamos a los individuos.

Para la creación de individuos probamos con distintas configuraciones iniciales, eligiendo al azar el número de capas ocultas, cada capa cuántas neuronas y sus respectivas funciones de activación.

En este apartado hubo algún problema con el número de neuronas de la primera capa, daba errores por ser demasiado grande en comparación con la de entrada, lo solucionamos limitando el número de neuronas en la primera capa.

La evaluación de un individuo se realiza construyendo el modelo secuencialmente, al comienzo se coloca la capa de entrada, construimos el modelo iterando por los genes y para finalizar la capa de salida con una función de activación softmax, que es la que hemos encontrado como mejor para datos de salida en One-Hot encoding.

Ahora toca entrenar el individuo con los datos de entrenamiento, probamos distintas configuraciones, haciendo uso de algunos atributos de los que disponemos en Tensor Flow, los explico a continuación:

**Epoch**

Un epoch es una “pasada” por todos los datos. Una iteración completa.

**Paciencia**

La paciencia permite parar el entrenamiento si el modelo se estanca en varios epoch

**Batch size**

Batch size es el número de datos que pasan hasta actualizar el modelo.

Hemos calculado los valores para que dependan de la generación en la que estamos y en el tamaño del modelo. Por ejemplo, un modelo más grande tarda más en converger, por lo que es mejor tenerle un poco más de paciencia.

Además con el paso de las generaciones, queremos evaluar qué tal se comporta el modelo cuanto más lo entrenamos por lo que aumentamos el número de epochs también según el tamaño y la generación en la que estamos. También, cuanto más pequeño es el batch size, más “Ajuste fino” conseguimos por lo que lo reducimos según el paso de las generaciones.

Debemos explicar primero el fitness, que es la medida que se utiliza en cada individuo para evaluarlo y compararlo con el resto. En nuestro caso, el fitness es la accuracy obtenida por el modelo en su entrenamiento probado con los datos de validación.

La selección de los mejores individuos puede realizarse con distintas técnicas como por ejemplo torneo, nosotros la hemos realizado con elitismo, esto consiste en elegir a los N mejores de cada generación.

Una vez hemos elegido a los mejores, los tenemos que “cruzar”, esto consiste en elegir los genes de uno y combinarlos con los de otro. Se puede hacer de varias formas, por ejemplo al comienzo probamos con escoger una proporción de cada uno random y después las juntamos, también probamos a elegir cada capa random, si la de uno y otro como máximo hasta la longitud, aunque al final nos terminamos quedando con esta última pero si alguno de los dos padres es más grande que el otro, se completan los genes restantes hasta tener la longitud del padre más largo, pues nos dimos cuenta que a mayor longitud, mejores resultados obtenemos.

Por último las mutaciones, esto consiste en que una vez generado el nuevo individuo por el cruce, introducimos la posibilidad de que exista una mutación, esto puede ser modificar una capa, cambiando el número de neuronas o la función de activación, quitar una capa o introducir una nueva con valores aleatorios.

#### Ejecución del algoritmo

Esta parte ya consiste solo en ejecutar las funciones que hemos creado, con algunos apuntes.

Para el algoritmo genético hay que decidir, cuántas generaciones se van a crear y el número de individuos que las componen. Normalmente en modelos complejos se necesitan muchas generaciones para obtener un buen resultado, pero en este caso, al tratarse de un entrenamiento de modelos, el tiempo empleado es muy grande, por lo que hemos tenido que hacer algunos ajustes.

Hemos hecho muchas pruebas probando estos números, por ejemplo, hemos elegido que las poblaciones sean de 20 individuos y 10 generaciones (20 \* 10 modelos en total), y aún siendo pocos, tarda poco menos de 1 hora en ejecutarse en una RTX 3060 con 4 hilos trabajando en paralelo para los entrenamientos.

Hemos probado con bastantes más combinaciones, probando si convenía aumentar el número de generaciones o el número de individuos por cada una. Un mayor número de generaciones permite afinar más en la solución, en cambio mayor número de individuos nos permite explorar más el espacio de soluciones.

En nuestro caso, debido al problema de la cantidad de cómputo hemos tenido que limitar las dos. Dado que no somos expertos en esta área y queríamos entrenamientos más cortos que nos permitan hacer pruebas y en hacer que la siguiente ejecución sea más prometedora que la anterior.

Otra parte importante es la elección de la siguiente generación, los siguiente individuos se forman a partir de la generación anterior, elegidos por elitismo, esto es, se seleccionan a los ‘k’ mejores de la actual y se cruzan entre ellos. En nuestro caso también estuvimos haciendo pruebas en esto y resulta que además de cruzarlos entre ellos, cruzarlos con el mejor hasta el momento según una probabilidad, nos daba mejores resultados -al menos a corto plazo-. Esto es debido a que cada vez que entrenamos una red neuronal, este se inicia con pesos aleatorios, esto hace que alguna veces converja mejor que otras, al probar tantas veces variaciones del mejor modelo, además de otras, nos aseguramos de que la solución converge bien y no se atasca en mínimos locales con tanta facilidad.

Al final de las generaciones obtenemos un modelo final, que al final de todas las generaciones obtuvo el mejor fitness.

#### Entrenando el modelo final

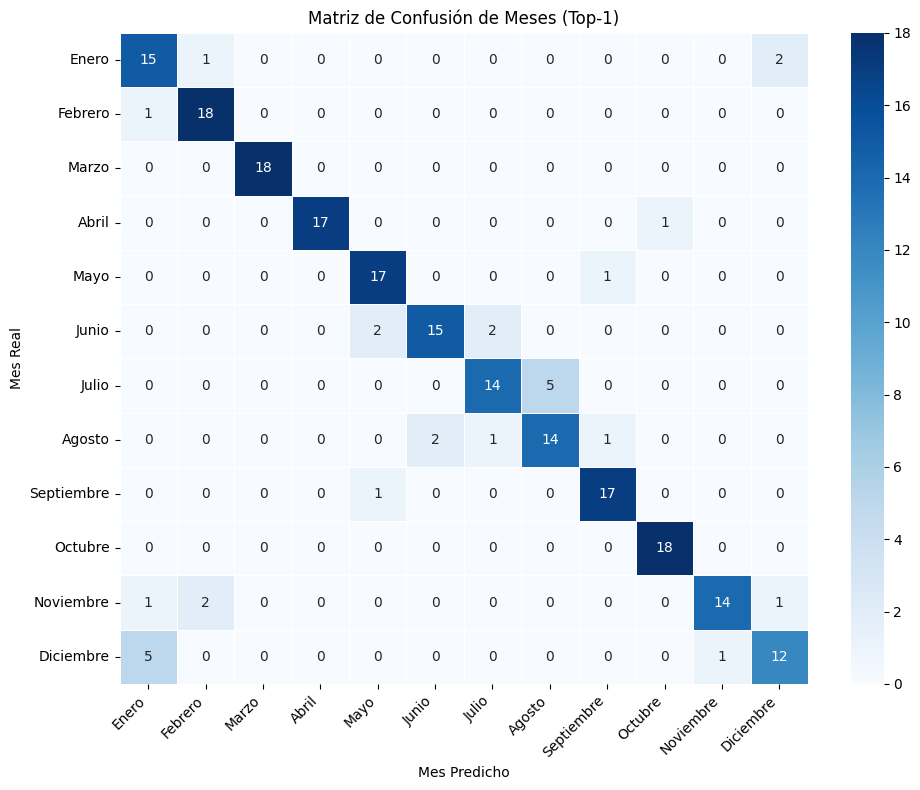
Ahora que ya tenemos la mejor arquitectura, hacemos el entrenamiento final, un entrenamiento más largo para darle tiempo a que converja mejor y así obtener un mejor resultado.

Este resultado lo comparamos con un modelo creado a mano, un modelo embudo que resulta no ser tan bueno como veremos ahora en los resultados.

### Precisión

|  | Top 1 | Top 2 | Top 3 |
| --- | --- | --- | --- |
| Precisión | 86.30% | 96.35% | 99.54% |

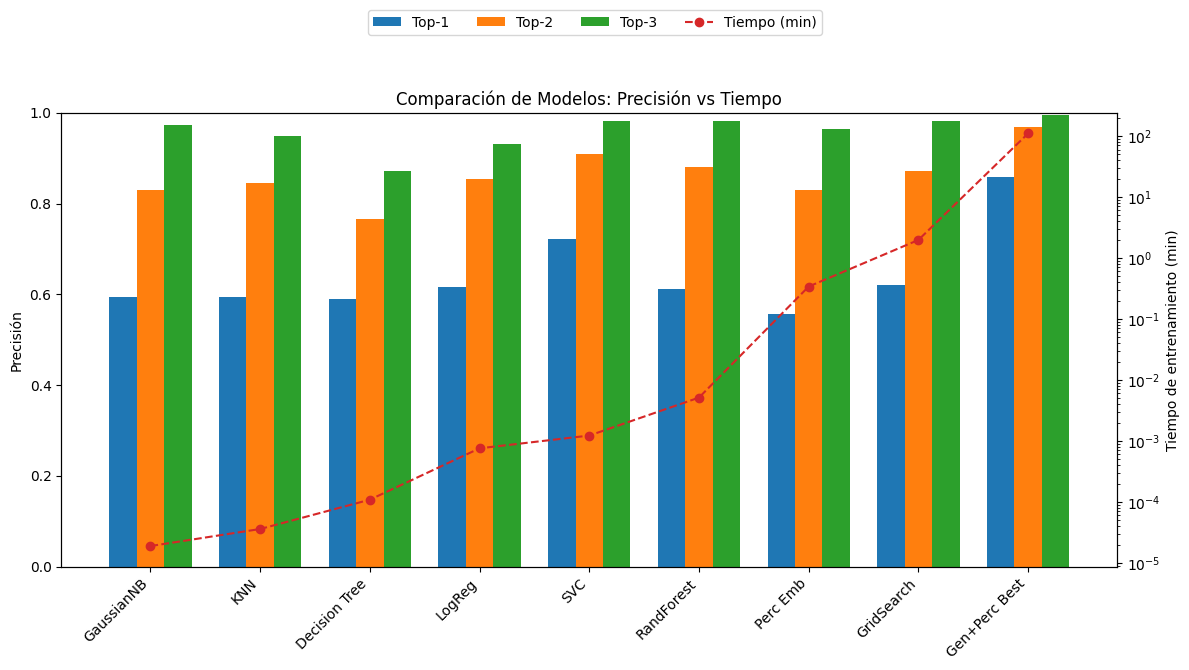
### Matriz de confusión



# Resultados

### Análisis

Para interpretar los resultados, queremos resaltar algunas observaciones al observar la gráfica comparativa:



En el gráfico podemos observar todos los modelos presentados anteriormente, también hemos añadido uno más, un perceptrón con forma de embudo, creado manualmente. Se muestra una comparación de las precisiones de cada modelo su Top 1, Top 2 y Top 3, que representan si el mes real estaba dentro de sus (1, 2, 3) primeras opciones.

Hemos incluido además una comparativa temporal, pues hay que destacar que no necesariamente con la mayoría de modelos cuanto más tiempo de computación mejor, excepto en el último caso, del que seguido discutiremos.

En la tabla la escala es logarítmica, y hemos ordenado los modelos de menos tiempo en entrenar a más.

### Analizando los valores de la imagen

En cuanto al top 3 encontramos que todos los modelos obtienen un número alto de aciertos, superando todos el 90% de aciertos.

Al pasar al top 2, observamos que la precisión baja un 5% de media, esto es un buen indicativo pues como hemos mencionado, dada la variabilidad del clima, este comportamiento es esperable.

La mayor diferencia la encontramos al observar el top 1. En este caso casi todos los modelos están en una precisión entre 55% y 72% sin incluir al mejor modelo, el perceptrón optimizado con genéticos.

El perceptrón con genético, destaca por mucho. No solo aumenta en las tres estadísticas el porcentaje si no que la mayor diferencia la encontramos en el top 1, consiguiendo un sólido 86,30%. Que es un 14,15% más que el siguiente mejor, el Support Vector Classifier con 72,15%.

Hemos incluido el perceptrón sin genético para poder hacer la comparativa con el genético. Es cierto que no hemos hecho muchas pruebas para encontrar una buena configuración para la red neuronal, pero era un poco la idea, en vez de encontrar a mano la mejor configuración, dejar que sea el genético el que encuentre una mejor solución.

Para contraste de los resultados, el modelo de embudo tiene una precisión de:

- Top-1 accuracy: 55.71%

- Top-2 accuracy: 83.11%

- Top-3 accuracy: 96.35%

Como se puede observar, es un resultado muy pobre, aunque como hemos visto, con mucho potencial de mejora.

### Conclusiones

Por lo tanto, nos queda claro que el mejor modelo, con diferencia, es la combinación del algoritmo genético junto con el perceptrón. Es un salto de precisión muy importante, llegando a un rendimiento muy superior al resto.

El único inconveniente es el tiempo y coste computacional, como se puede observar el tiempo para entrenar al genético+perceptrón es mucho más grande que el resto, incluso solo el perceptrón, pues tardan varios órdenes de magnitud más.

Por lo que en caso de que se quiera obtener un modelo con un buen rendimiento pero no se disponga de los recursos para entrenar un modelo, podríamos usar el Support Vector Machine, el cual tiene un coste ínfimo en comparación, con unos resultados aceptables en la proporción precisión/tiempo.

# Bibliografía

[Climatología. Precipitaciones](https://www.madrid.org/iestadis/fijas/coyuntu/otros/clpreci.htm)