Reporte Evaluación 2

En este documento se describen los experimentos realizados para comparar 2 técnicas de vectorización (TF-IDF y Word Embeddings) en el entrenamiento de redes neuronales artificiales hechas con PyTorch para el reconocimiento y clasificación de textos. Así mismo se comparan algunas arquitecturas para las redes neuronales y algunos hiper parámetros.

Word Embeddings

Word Embedding es una técnica de representación de los textos que consiste en mapear cada palabra a un vector numérico, esto tiene la consecuencia de que es posible representar frases completas con un solo vector de N dimensiones. Comúnmente este mapeo se realiza con una red neuronal tradicional ya entrenada y disponible para su uso. En esta práctica se usará FastText, un clasificador de texto que cuenta con varios modelos entrenados para distintos idiomas y variaciones del español.

```
def vectorizar_embeddings(X, Y, variante: str, test_size=0.2, val_size=0.1):
    if variante == 'MX':
        ft = fasttext.load_model('./fasttext/MX.bin')
    elif variante == 'ES':
        ft = fasttext.load_model('./fasttext/ES.bin')
    elif variante == 'GEN':
        ft = fasttext.load_model('./fasttext/cc.es.300.bin')

    X_ = X.map(lambda x : ft.get_sentence_vector(x)) # type: ignore
    X_ = np.vstack(X_.to_numpy())
    X_train, X_test, Y_train, Y_test =  train_test_split(X_, Y, test_size=test_size,
stratify=Y, random_state=RANDOM_STATE)
    X_train, X_val, Y_train, Y_val =  train_test_split(X_train, Y_train,
test_size=val_size, stratify=Y_train, random_state=RANDOM_STATE)

    return X_train, X_test, X_val, Y_train, Y_test, Y_val
```

La principal ventaja de word embeddings sobre otros métodos de vectorización como TF es que no hace falta preprocesar el texto, dado que el modelo ya está entrenado para reconocer los tokens.

Codificación One Hot

Para la clasificación usando redes neuronales es común representar cada clase con un número, por ejemplo si se trata de clasificación de animales 1 representa gato, 2 perro y 3 ratón. Al tener más de 2 clases posibles surge un problema relacionado con el orden, pues si la representación de cada clase es numérica, el modelo puede aprender erróneamente que 1 < 2 < 3, en este caso gato < perro < ratón, lo que puede sesgar los resultados. Para evitar este sesgo existe la codificación One Hot, consiste en representar cada clase con un vector en lugar de un número concreto, este vector se forma con la misma cantidad de columnas que de clases y será un vector con 0 en todas las columnas menos la que representa la clase predicha. En el ejemplo anterior en lugar de representar gato con el valor 1 se representa con el vector [1, 0, 0], perro con [0, 1, 0] y ratón con [0, 0, 1].

```
def torchificar(X: list, Y: list, representacion: str, one_hot_encoding=False):
    le = LabelEncoder()
    X_torch = []
```

```
    Y_torch = []

    for x in X:
        if representacion == 'TF-IDF':
            x_np = x.toarray().astype(np.float32)
        elif representacion == 'Embeddings':
            x_np = x.astype(np.float32)
        x_torch = torch.from_numpy(x_np) # type: ignore
        if torch.cuda.is_available():
            x_torch = x_torch.cuda()
        X_torch.append(x_torch)

    for y in Y:
        y_vec = le.fit_transform(y)
        y_torch = torch.from_numpy(y_vec)
        if torch.cuda.is_available():
            y_torch = y_torch.cuda()
        Y_torch.append(y_torch)
    if one_hot_encoding:
        Y_torch[0] = one_hot(Y_torch[0], num_classes=NUM_CLASES).float()

    return X_torch, Y_torch
```

Arquitectura

Para todos los experimentos se usará la clase FeedForwardNeuralNetwork

```
class FeedForwardNeuralNetwork(nn.Module):
    def __init__(self, input_size: int, hiden_sizes: list[int], output_size: int):
        super().__init__()
        self.fcl = nn.ModuleList()
        self.act = nn.ModuleList()

        # Las capas se forman por pares de numeros, en total 1 par menos que la lista
neuronas
        neuronas = [input_size] + hiden_sizes + [output_size]
        for i in range(len(neuronas) - 1):
            self.fcl.append(nn.Linear(neuronas[i], neuronas[i + 1]))
            nn.init.xavier_uniform_(self.fcl[i].weight) # type: ignore
            nn.init.zeros_(self.fcl[i].bias) # type: ignore
        #Al aplicar CrossEntropy se necesita una función de activación menos que de
capas
        for i in range(len(neuronas) - 2):
            self.act.append(nn.ReLU())

    def forward(self, X):
        x = X
        for i in range(len(self.act)):
            x = self.fcl[i](x)
            x = self.act[i](x)
        x = self.fcl[-1](x)     #No activar la última capa
```

```
        return x
```

Esta clase permite una cantidad arbitraria de capas con un número de neuronas arbitrario, ya que en todos los experimentos se usará CrossEntropy como la medida para el error, es necesario no activar la última capa durante el forward pass. La función de activación utilizada fue ReLu y el método de optimización fue Adam.

Experimentos

Para comparar los métodos de vectorización, cantidad de capas en la arquitectura y otros hiper parámetros se realizaron experimentos combinando las variables como se indica en la siguiente tabla.

Características de los experimentos

Variables	Valores
Vectorización	• TF-IDF • Word Embeddings
Variante del español (Solo Word Embeddings)	• Español (General) • Español de España • Español de México
Arquitectura de capas ocultas	• 3 capas (256, 128, 64) • 4 capas (512, 256, 128, 64) • 5 capas (1024, 256, 128, 64)
Learning Rate	• 0.1 • 0.01 • 0.001
Batch Size	• 32 • 16
Constantes	**Valores**
Método de Optimización	Adam
Función de pérdida	CrossEntropyLoss
Función de activación	ReLu
Métricas de desempeño	F1-score Accuracy Precision Recall
Épocas	100 (Sin early stop)
N-gramas (Sólo TF-IDF)	Unigramas y Bigramas
Preprocesamiento del texto (Sólo TF-IDF)	Normalización, eliminar Stopwords y Stemming

Los valores de Learning rate, batch size y neuronas en la última capa oculta fueron tomados de los mejores modelos resultantes de la Práctica 2. Solo se agregó un valor extra de learning rate debido al uso de Word Embeddings

Código para TF-IDF

```
if EXPERIMENTAR:
   vec = 'TF-IDF'
   n_modelo = 1
   X, Y = leer_datos(filename, transform=True)
   X_train, X_test, X_val, Y_train, Y_test, Y_val = vectorizar_TF_IDF(X, Y)
   Xs, Ys = torchificar([X_train, X_test, X_val], [Y_train, Y_test, Y_val], vec,
one_hot_encoding=True) # type: ignore

   for arq in arquitecturas:
       for lr in learning_rates:
           for b in batch_sizes:
               caracteristicas = {
                   'ID':n_modelo,
                   'Vectorizacion':vec,
                   'Variante':None,
                   'Entradas':Xs[0].shape[1],
                   'Arquitectura':arq,
                   'Salidas':NUM_CLASES,
                   'LearningRate':lr,
                   'BatchSize':b,
                   'CapasOcultas':len(arq)
               }
               modelo = FeedForwardNeuralNetwork(Xs[0].shape[1], arq, NUM_CLASES)
               funcion_perdida = nn.CrossEntropyLoss()
               optimizador = optim.Adam(modelo.parameters(), lr=lr)
               historial_perdida = entrenar(Xs, Ys, modelo, optimizador,
funcion_perdida, epocas=epocas, batch_size=b)
               f1, a, p, r = evaluar(Xs[1], Ys[1], modelo, verbose=True)
               caracteristicas.update({
                   'Epocas':epocas,
                   'F1-score':f1,
                   'Accuracy':a,
                   'Precision':p,
                   'Recall':r,
                   'Historial':historial_perdida
               })
               guardar_modelo(modelo, caracteristicas)
               n_modelo += 1
```

Código para Word Embeddings

```
if EXPERIMENTAR:
   vec = 'Embeddings'
   X, Y = leer_datos(filename)
   for v in variantes_esp:
       X_train, X_test, X_val, Y_train, Y_test, Y_val = vectorizar_embeddings(X, Y, v)
       Xs, Ys = torchificar([X_train, X_test, X_val], [Y_train, Y_test, Y_val], vec,
one_hot_encoding=True) # type: ignore
       for arq in arquitecturas:
           for lr in learning_rates:
               for b in batch_sizes:
                   ...
```

Resultados

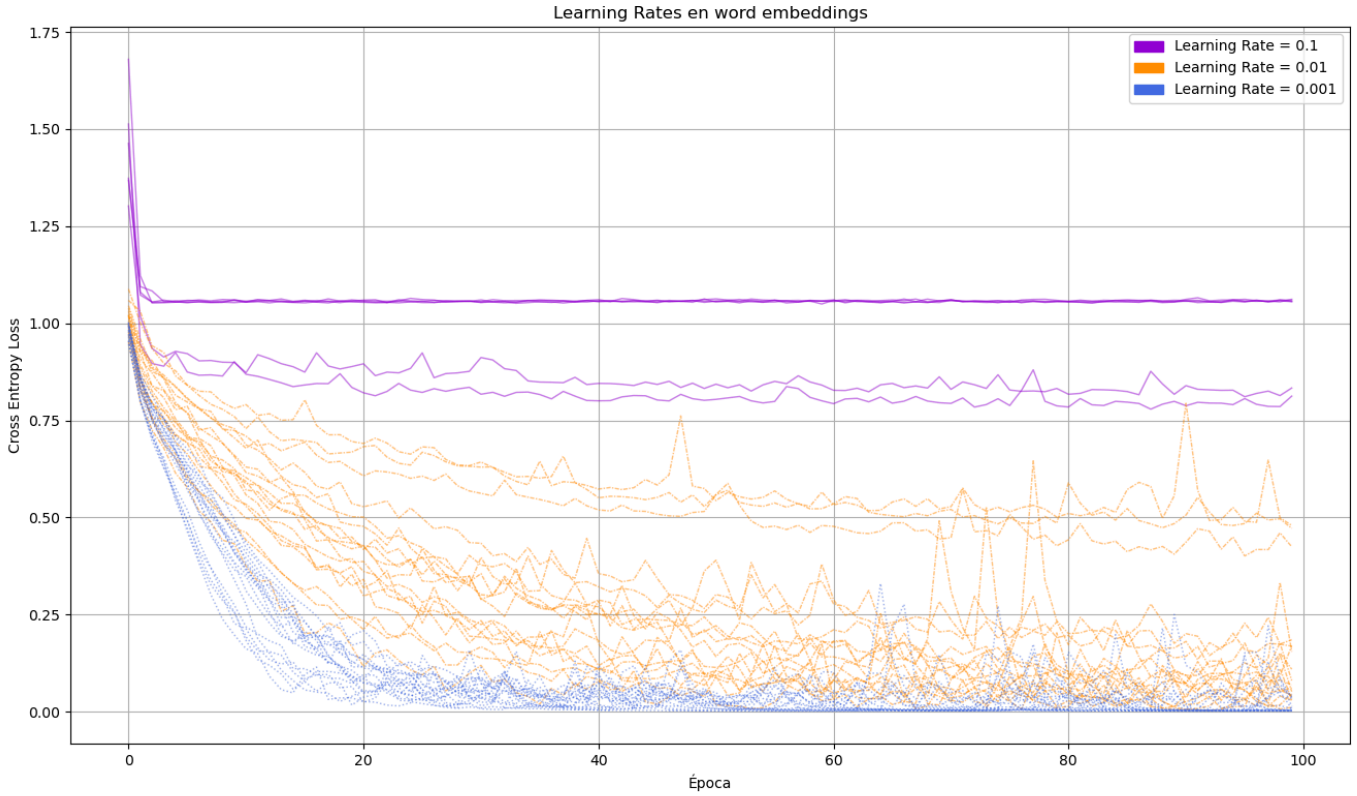
En la siguiente sección se muestran gráficos para comparar los distintos valores de los experimentos. Nótese que en las gráficas fueron excluidos algunos modelos por tener valores de error mucho más grandes que el resto, solo con el propósito de mejorar la visualización de los resultados.

```
def graficar_conjunto(listas_modelos, nombres_labels, titulo, filename, epocas,
opacidad=0.2, error_max=2):
   colores = ['black', 'darkred', 'forestgreen', 'darkviolet', 'darkorange',
'royalblue']
   marcadores = ['solid', (0, (3, 1, 1, 1)), 'dotted', (0, (5, 1)), 'dashed', (0, (5,
10)), (5, (10, 3)), (0, (3, 5, 1, 5))]
   i = 0
   offset = len(colores) - len(listas_modelos)
   patches = []
   plt.figure(figsize=(16, 9))
   plt.title(titulo)
   n = 0
   for lista in listas_modelos:
       color = colores[i + offset]
       for diccionario in lista:
           if np.array(diccionario['Historial']).max() > error_max:
               n += 1
           else:
               plt.plot(np.arange(epocas), diccionario['Historial'][:epocas],
color=color, linestyle=marcadores[i], linewidth=1, alpha=opacidad)
       patches.append(mpatches.Patch(color=color, label=nombres_labels[i]))
       i += 1
   plt.figtext(0.13, 0.03, f'*{n} modelos excluidos para mejorar la visualización')
   plt.legend(handles=patches)
   plt.xlabel('Época')
   plt.ylabel('Cross Entropy Loss')
   plt.grid(True)
   plt.savefig(filename)
   plt.close()
def graficar_metricas(datos, titulo, filename: str):
   nombres_metricas = ['F1-score', 'Accuracy', 'Precision', 'Recall']
   _, ax = plt.subplots(figsize=(16, 9))
   width = 0.2
   grupos = np.arange(len(datos[0]))
   i = 0
   for metrica_p in datos:
       offset = width * i
       rects = ax.bar(grupos + offset, metrica_p, width, label=nombres_metricas[i])
       ax.bar_label(rects, padding=3)
       i += 1
   ax.set_xticks(grupos + 3 * width / 2, ['TF-IDF', 'Embeddings'])
   ax.set_ylabel('Puntuación')
   ax.set_title(titulo)
   ax.legend()
   plt.savefig(filename)
```

```
plt.close()
```

Learning Rate

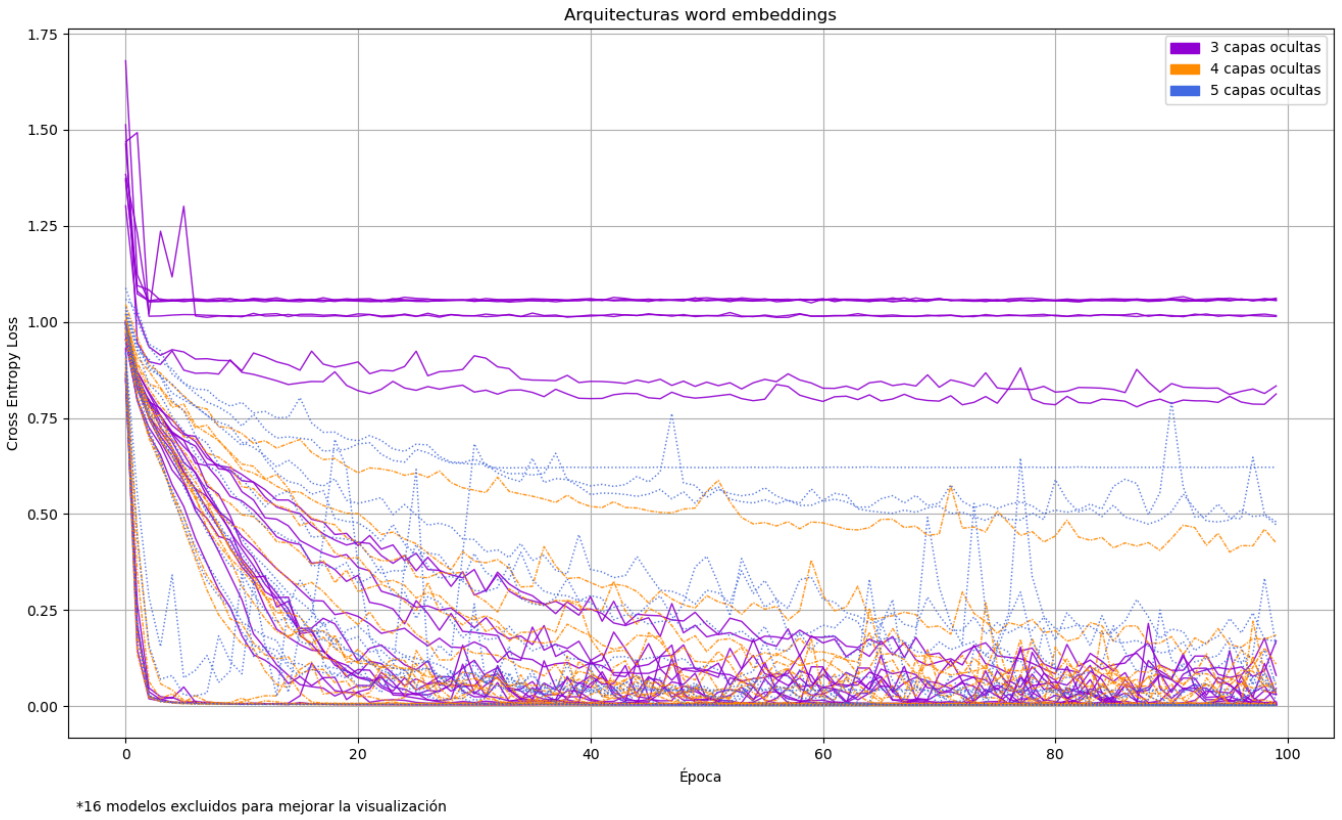
Usando el algoritmo de Adam es posible observar como los 2 learning rates que fueron mejores en los experimentos anteriores, ahora son notablemente peores, es por ello que se agregó a los experimentos un learning rate aún menor. Adam funciona mucho mejor con learning rates pequeños.



*12 modelos excluidos para mejorar la visualización

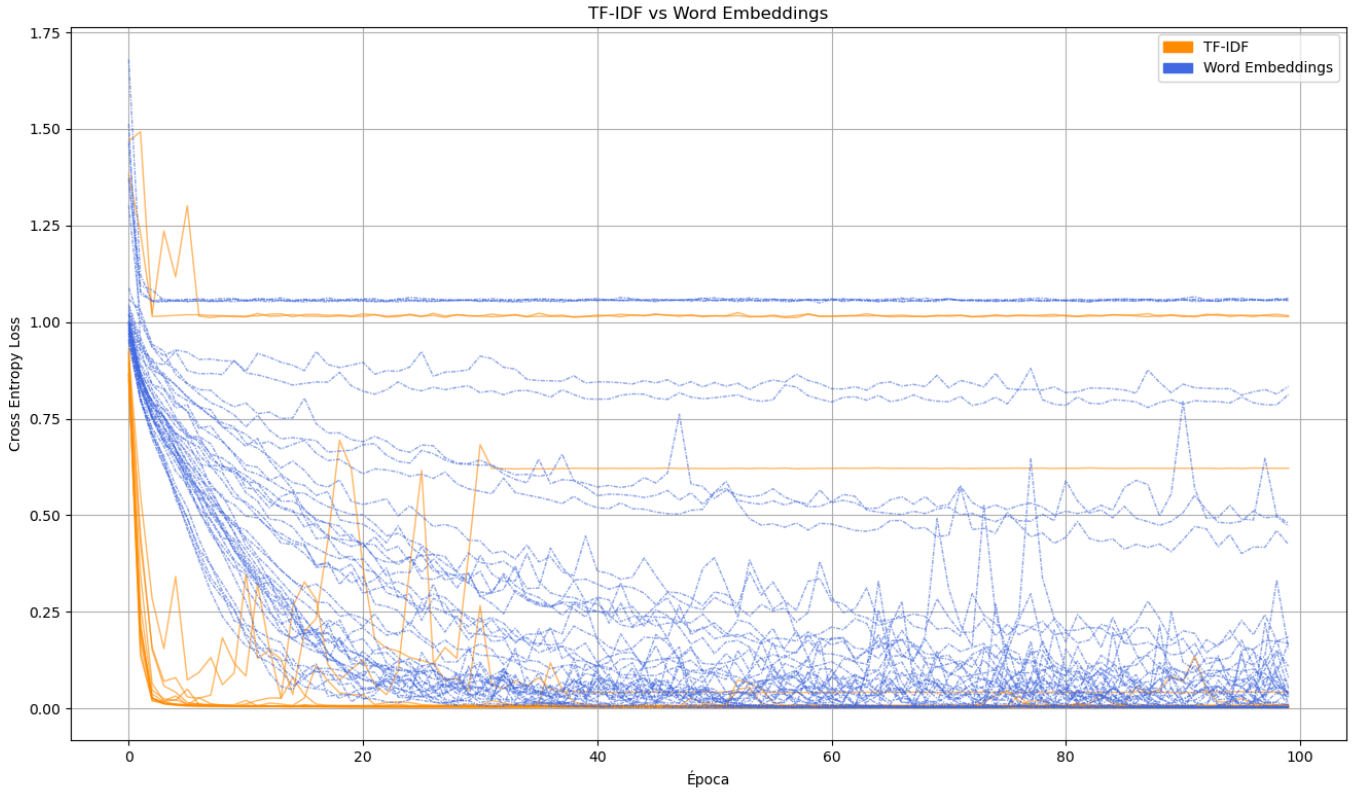
Arquitectura

No se observó alguna diferencia al usar las distintas arquitecturas, más experimentación debe ser realizada, cambiando la cantidad de neuronas y/o las funciones de activación.

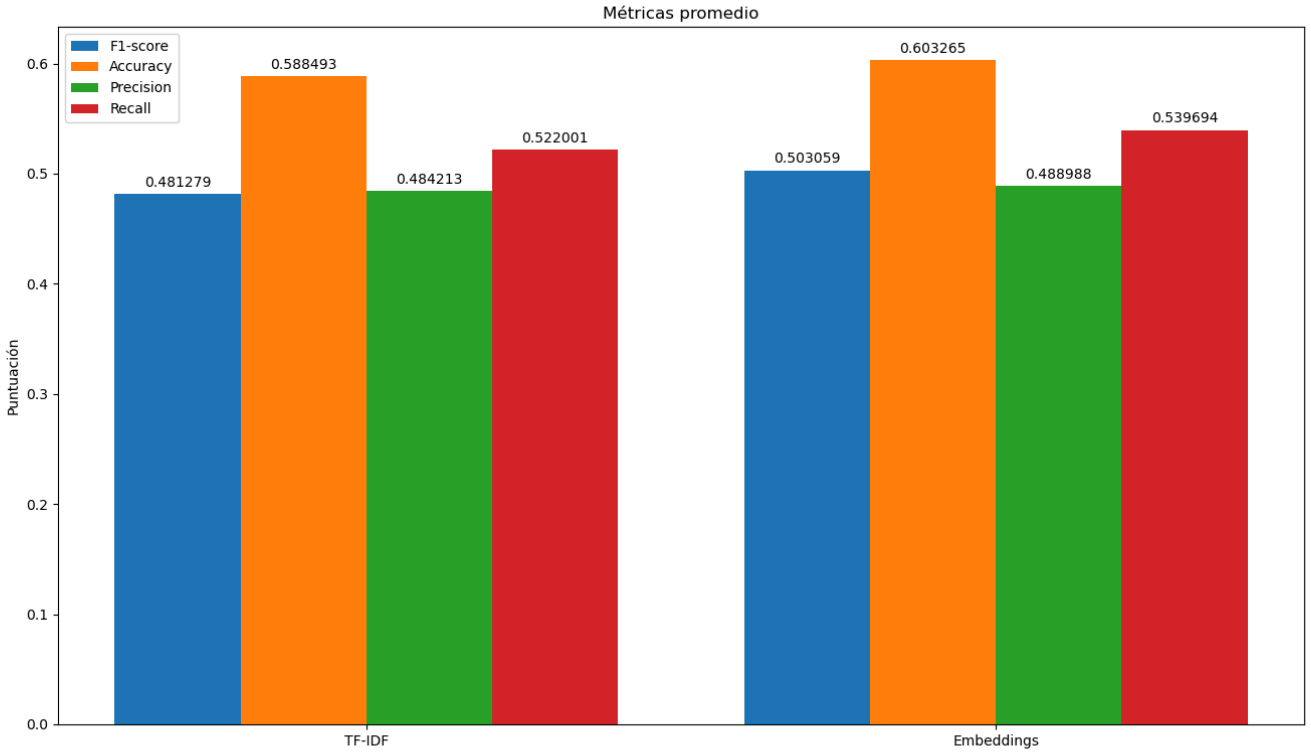


Vectorización

Finalmente se obtuvieron sorprendentes resultados al comparar TF-IDF con word embeddings. Primero se observa claramente como la vectorización tradicional de TF-IDF converge más rápido en prácticamente todos los casos, sin embargo esto contrasta con los resultados promedio de las métricas tomadas, pues word embeddings obtuvo ligeramente mejores resultados en promedio a pesar de converger más lento. Esto indica la posibilidad de que word embeddings sea mejor que TF-IDF si se contempla un entrenamiento más largo, lo cual es posible fácilmente dado que word embeddings tiene solo 300 dimensiones mientras que TF-IDF tiene más de 18 mil, por lo que el incrementar el número de épocas no representa ningún problema.



*16 modelos excluidos para mejorar la visualización



Conclusiones

Word embeddings obtuvo modelos de mejor calidad en promedio que TF-IDF y esta parece mejorable incrementando la cantidad de épocas durante el entrenamiento de los modelos, lo que se complementa con la menor dimensionalidad de esta forma de vectorización. De igual forma TF-IDF presentó una convergencia mucho más rápida, este hecho puede ser útil para problemas específicos.

El algoritmo de optimización Adam es un algoritmo poderoso que acelera la convergencia de los modelos, sin embargo requiere de learning rates pequeños, pues valores grandes como 0.1 pueden resultar inestables o con convergencia en errores grandes y modelos de muy mala calidad.

La cantidad de capas ocultas en las redes neuronales no tuvieron un efecto notable en la calidad de los modelos, no al menos con una progresión de dividir a la mitad el número de neuronas entre cada capa hasta un límite de 64 neuronas. Se requiere de más experimentación con otra cantidad de capas y distintas progresiones para llegar a resultados concluyentes.