LDSI W2021 Project Report

Name: Batuhan Erden
Gloss ID: tum_ldsi_18

Summary

In this experimental project work, a BVA decision search engine was developed, which eventually showed very promising performance results in processing decisions and inferring case outcomes. After a balanced partitioning of the data, three different segmentation techniques were analyzed on the training set, and this error analysis provided really good insights into which segmenter is best suited for specific tasks. Using Savelka's law-specific segmenter and Spacy's great built-in features, 10k unlabeled BVA decisions were tokenized, which were then fed into a 100-dimensional embedding model trained with FastText, where the neighborhood relations generated were accurate and the semantic meaning between neighbors is well established in the legal context. All this allowed training a linear classifier (Linear Support Vector Machine) and a non-linear classifier (Random Forest) using TFIDF and Word Embedding Featurization techniques. In the end, the best trained classifier was able to correctly classify most of the types, although there is still much room for improvement as a future work. In the following sections, each of the processes in the pipeline for creating a BVA decision classifier is explained in detail to justify the choices made.

Dataset Splitting

The first requirement is to extract only the annotated documents. Analysis of the data shows that out of 540 documents, only 141 are annotated. After removing the unannotated documents, the data needed to be sampled in a balanced way. That is, there must be an equal number of approved and denied decisions. The data set is divided into three smaller sets, with 80% of the documents in the training set and 10% each in the development set and test set. For example, in the test set there are 7 approved decisions and 7 denied decisions (14 cases in total).

IDs of the documents in the development set:
'61aea55d97ad59b4cfc412d2', '61aea55e97ad59b4cfc412d7', '61aea57097ad59b4cfc41365', '61aea55e97ad59b4cfc412e6', '61aea57397ad59b4cfc413a7', '61aea55e97ad59b4cfc412d4', '61aea57397ad59b4cfc41393', '61aea55f97ad59b4cfc41322', '61aea55f97ad59b4cfc41328', '61aea57497ad59b4cfc413e7', '61aea57497ad59b4cfc413c4', '61aea57497ad59b4cfc413b2', '61aea55e97ad59b4cfc412fd', '61aea57297ad59b4cfc41382'

IDs of the documents in the test set:
'61aea55c97ad59b4cfc41299', '61aea55f97ad59b4cfc41307', '61aea57097ad59b4cfc41355', '61aea57097ad59b4cfc4135a', '61aea55f97ad59b4cfc41334', '61aea55f97ad59b4cfc4131a', '61aea55c97ad59b4cfc412af', '61aea56f97ad59b4cfc41342', '61aea55f97ad59b4cfc41301', '61aea57097ad59b4cfc41361', '61aea57097ad59b4cfc4135e', '61aea55d97ad59b4cfc412bf', '61aea57497ad59b4cfc413bd', '61aea57497ad59b4cfc413be'

Sentence Segmentation

Now that the dataset is partitioned, the next step is segmentation analysis, where three different techniques are analyzed: Standard and Improved Segmentation with *Spacy* and *Savelka's Law-Specific Sentence Segmenter*. The segmentation process plays a major role in the success of the model, as this is where the best technique gets selected that will then be used to sentence-segment decision and even tokenize those split sentences. The error analysis performed to select the best segmenter is nothing more than the calculation of *precision*, *recall* and *F1* for the entire training data set. A split is said to be "*matched*" if the distance between the split and the true set is at most 3 characters.

First, a very powerful sentence segmenter, *Spacy*, is implemented and applied to the training set without worrying about any special cases. Although *Spacy* is considered very powerful in the *NLP* world, it did not perform so well because the corpus contains legal texts that require many special segmentations. However, as can be seen from the error analysis below, the problems are not unsolvable, and most of them can be fixed with the extension functions built into *Spacy*.

Error Analysis	Average Precision	Average Recall	Average F1 Score
Comparing both indices	45.94%	59.27%	51.76%

Table 1: Error Metrics for Spacy Segmentation

Although the scores in the table above are rather low, they can be improved by relaxing the matching condition. That is, instead of comparing the characters at both the beginning and the end of the splits, only the initial indices can also be compared, resulting in a higher score. In the last parts of this section, the relaxed error analysis is also performed to obtain more realistic results. Before turning to the improved version of *Spacy*, let's analyze the *over-segmented* and *under-segmented* instances by looking at the three documents with the worst *average F1* scores. The most recurring types for these 3 documents were found to be *Citation*, *EvidenceBasedOrIntermediateFinding*, *EvidenceBasedReasoning*, *Evidence*, and *LegalRule*, with *Citation* being the most recurring. From the knowledge gained in this course, it is easy to interpret that there are some unique identifiers for these types so that they are identified as they are. First of all, these types contain numbers, parentheses and dots. Looking at them more closely, it is obvious that some of the problematic tokens are **"Vet. App."** and **"CF. 38"** that occurs a lot in citation sentences and probably the reason why the worst performing documents have a lot of sentences annotated as citations. On the other hand, *Spacy* also tries to split the sentence due to the fact that the characters like **“)”, “,”** and **“;”** normally define sentence breaks when the tokens like **“(2004),”** and **“(2004);”** should not be split. Last but not least, some special tokens like **"DOCKET NO."** and **"), DATE))"** occur only once per document in the *CaseHeader* and will increase the error for each document. In short, one needs to handle the special cases for more accurate segmentation and also numbered lists like **1.**, **2.** and **3.** that causes sub-segmentation.

In the improved version of this segmenter, all the special cases described above but the numbered lists are handled by Spacy's built-in functions. In addition, some additional tokens like **Fed. Cir.**, **Fed. Reg.** and **Pub. L. No**. are added as non-splittable tokens. Also, the long underscore seen in the *CaseFooters* is added as a special case that will prove useful in the later steps such as tokenization. These extensions to the segmenter improved the performance by almost **10%** in terms of *F1* score as can be seen in the table below.

Error Analysis	Precision	Recall	F1 Score
Comparing both indices	57.00%	65.65%	61.02%

Table 2: Error Metrics for “Improved” Spacy Segmentation

The performance is still not really convincing. However, the next and last segmenter tested is the *Savelka’s law-specific sentence segmenter*, which is helpful in combating the over- or under-segmentation of legal texts as it was trained on legal documents [1]. As demonstrated in the table below, *Savelka’s segmenter* did really well in outperforming the *Spacy* by a resounding **92%** *F1*.

Error Analysis	Precision	Recall	F1 Score
Comparing both indices	74.54%	86.72%	80.17%
Comparing only the start indices	85.06%	98.97%	91.49%

Table 3: Error Metrics for Segmentation with Savelka's Law-specific Sentence Segmenter

However, there are still some errors. The first improvement made on top of this segmenter is to keep the *CaseHeader* intact by merging *the first 5 splits*, which surprisingly is not done by the segmenter itself as *Spacy* prevents this *under-segmentation*. Another error found while investigating the worst performing documents was the handling of special characters and numbers. For the document with **ID='61aea55d97ad59b4cfc412bc'** with lots of keywords like ***"HERTZ"***, most of the numbers, metrics, directions, and special abbreviations are split into different sentences when they should be in a different whole sentence. However, this is only the case for a minority of documents, and since *Savelka's law-specific segmenter* was able to sentence-segment decisions much better, even splitting headers, it will be used as a sentence segmenter from now on.

Preprocessing

Having chosen the best sentence segmenter for moving forward, now it is time to implement the tokenization as it will be later needed to train an embeddings model. The dataset utilized for this step is an unlabeled dataset where there are ***30,000*** unlabeled documents. First and foremost, using the *Savelka's law-specific sentence segmenter,* the ***30,000*** decisions in the unlabeled dataset are sentence-segmented, which resulted in **3,360,495** sentences. This process took around *2 hours* on *a system with 8 threads and enough memory*. Further analysis of the number of sentences in each document revealed that the average number of sentences in all unlabeled decisions was **112**, while the maximum number of sentences was **974**. As can be seen in *Figure 1* below, where the x-axis is the number of sentences and the y-axis is the number of documents, these statistical values hold and almost all documents are distributed around the mean, with most documents containing fewer than **200** sentences.

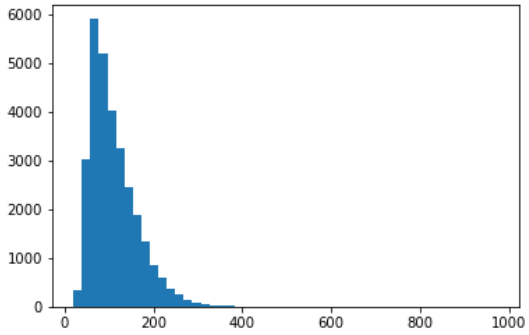

Figure 1: Number of sentences across all unlabeled decisions

Now that the decisions are sentence-segmented, the next step is to tokenize those sentences. Tokenization helps interpret the meaning of the text by analyzing the order of words [2]. The created tokenization function is used not only for training an embedding model, but also for preparing the feature vectors for the classification process. Therefore, it was a crucial task to try different sentence segmentation methods to learn possible occurrences of *under-segmentation* or *over-segmentation*. For further splitting the sentence-segmented decisions into tokens, the improved version of *Spacy* was utilized due to its great built-in functions allowing the definition of special cases. The fact that Spacy had a lot of *over-segmented* sentences in the above section is no longer a problem here, but rather an advantage, since the goal here is to get as many tokens as possible. Note that

tokenizing **3,360,495** sentences is a computationally-expensive process, which took around *7 hours* on *a system with 8 threads and enough memory*. Having it run in parallel helped, but it was still around *2 hours*.

As explained in the previous paragraph above, in the best case, *Spacy* provides as many splits as possible allowing each token to be a word, number or punctuation, which can be processed through some filters. Keep in mind that the same improved Spacy segmenter described above is used here to tokenize so that the tokens like **"Vet. App."** and **"CF. 38"** as they only have semantic meaning when seen together, and the long underscores in the *CaseFooters* are not split into different tokens. After the tokens are obtained using *Spacy*, each of them goes through some filtering processes with the help of the *pos_* attribute of the tokens storing whether the token is a number, a punctuation, a verb or a noun. The first filtering step is the removal of punctuation, spaces and particles, especially if the tokens consist only of them. This step is very important because it frees the data from uninformative noise that would otherwise make the data more complex and harder for the models to train on it. The next step is to identify and simplify the dates and replace them with a special token (**<DATE*X*>**) where ***X*** is the number of digits in the date string. For example, in tokenization, **1998** and **11/02/2006** are converted to **<DATE4>** and **<DATE10>** respectively. The main idea behind this conversion is to classify types, which usually contain many dates, easier. Moreover, the simplification is very important because the corpus contains too many different dates, such as *a unique identifier (ID)* in a database, which would make it difficult to train the model. On the other hand, the numbers that are not dates must also be simplified for the same simplification reason. In the next step, if the token is not a date and nevertheless a number, it is converted into a special token (**<NUM*X*>**) where ***X*** is the number of digits in the respective number. In the last step, if the token successfully has passed all of the checks above, it first gets converted to its lemma. Another advantage of *Spacy* is that it just does not split the sentence but also extracts the lemma from words with its built-in trained lemmatizer. Lemma is really a very strong prior, since it converts, for example, "***held***" to "***hold***" and "***commits*"** to "***commit***", and it does for all known English words with its trained English pipelines contained in the library [3]. The token's lemma goes through a final filtering process in which all characters are lowercased, since the distinction between upper and lower case usually has no semantic meaning, and the non-alphanumeric characters are also removed so that the uninformative noise tokens from the tokens can be removed. For instance, the token "*non-Federal*" becomes "*nonfederal*" at this step. Going through all these steps and analyzing the results, it was "*good enough*" to move forward as the majority of the tokens were meaningful.

Using the tokenization function described in detail above, all **3,360,495** sentences generated from the unlabeled corpus are tokenized, which produced **59,403,951** tokens in total. As can be seen in *Figure 2* below, where the x-axis is the number of tokens and the y-axis is the number of sentences, the distribution is similar to that of *Figure 1*, which was to be expected since the documents with more sentences should also have a higher token count. While the average number of tokens is **1,980**, the document with maximum tokens has **20,843** tokens. From *Figure 3* below, which shows the number of *the 25 most frequently occurring* tokens, it is also easy to see that stop word tokens such as "***the***", "***be***", "***of***" and "***and***" are very common. These tokens are not deleted, but their deletion could help the model since they have no semantic meaning. Last but not least, training an embedding model with FastText requires writing the tokens to a file in which each line consists of the tokens of a sentence separated by a single space as this is how *FastText* expects an input. In the final step, the generated tokens are written to a file after randomizing the order of the sentences. It is important to note that only the records that have *at least 5 tokens* are written to the file, resulting in only the tokens from **2,646,783** out of **3,360,495** sentences being written to the file.

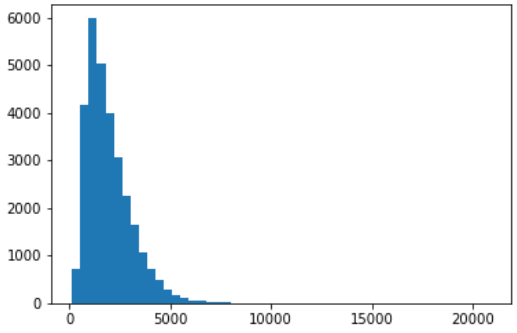

Figure 2: Number of tokens in each sentence

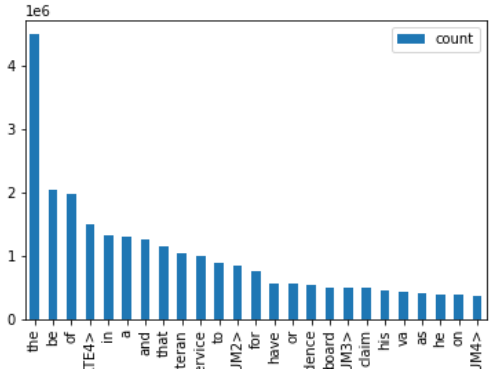

Figure 3: 25 tokens with the highest frequency

Custom Embeddings

Using the tokens written to a file, an embedding model with FastText is trained. The parameters are set so that the model produces a *100-dimensional vector model* with *standard character N-gram parameters* and *a minimum number of word occurrences* of **3**. Some parameters for *the minimum number of word occurrences* were also tried, but then **3** was chosen because it performed best in the classification process discussed below. With *a minimum word occurrence count* of **3**, the number of words was **27,734**, while it was **12,765** for *a minimum word occurrence count* of **20**. Using neighborhood analysis, the best number of epochs for training the model was set to **20**. With this number of epochs, *a machine with 8 threads* required almost half an hour to train **61 million** words. As a result, training the embeddings model with the above parameters resulted in a vocabulary of size **27,734**.

The performance of the model can be analyzed by evaluating the custom embeddings manually. Analyzing the *nearest neighbors* of some popular words in *BVA decisions* can help to understand the patterns in the embeddings and how the model makes the connection between closer words. The *10 nearest neighbors* of the following strings were analyzed. The following analysis showed that the embeddings model was "good enough" and could be used for further steps.

1. "***veteran***": The neighboring string are like "***his***", "***he***", "***furthermore***", "***moreover***", which makes really sense as the context of this particular string is used mostly with the pronouns "***he***" and "***his***" or usually it follows with what happened to the veteran. Hence, the string "***furthermore***", "***moreover***" and "***moreover***" really makes sense here. For example, in one BVA decision, we have the text "Thus, there is no medical evidence that shows that the veteran suffered from hepatitis during service.", which can semantically mean that the veteran had no medical evidence to support his claim, **furthermore**, it cannot be decided that he has suffered from hepatitis.
2. "***v.***": Meaning "**versus**" in the legal context. Among the neighbors, one can see areas like "**brainerd**" and "**jacksonville**", which can mean in the legal texts that it is the Veteran versus Brainerd or Jackonville. The string is also associated with some disorders like "**chlamydiazyme**", "**chlamydia**" and "**urosepsis**", for which I could not find a satisfying explanation.
3. "**argue/argues**": For this string, I am a little surprised I did not see "**that**" in the neighbors, since usually it is common to see "**that**" after "**argue**". This is obviously a good thing since "**that**" does not have a semantical meaning most of the time. On the other hand, strings like "**assert**", "**criticize**", "**dispute**" and "**agree**" are some of the closest neighbors, which makes sense because when something is argued in a BVA decision, it is usually to **assert** or **criticize** a point or **agree** with a decision.
4. "**ptsd**": This string is a disorder and it is successfully associated with other disorder-related words like "**psychiatric**", "**anxiety**", "**bipolar**", "**stressor**",

"**dysthymia**". It is also amazing that it successfully identified a typo "**pstd**" as it is the second nearest neighbor, from which one can deduce that such typos are also classified.

5. "**grant/grants**": It has the nearest neighbors "**entitlement**", "**warrant**" and "**award**" as these are granted in *BVA decisions* like a claim for the entitlement being awarded. Also, some of the closest terms are "**connection**" and "**service**," which is really correct because one can analyze many examples where "**grant**" occurs with "**connection**" and "**service**", for example, in one of the *BVA decisions* it is stated that **service connection** can be **granted** even if one has a disability.
6. "**korea**": The connection is successfully made with the neighbors as the closest ones are also countries, such as "**germany**", "**vietnam**", "**panama**" and "**seoul**", and also strings like "**demilitarize**", which means removing all forces from an area.
7. "**holding**": The most interesting one is the "**ruling**" neighbor as some citation sentences talk about the **holding** of a **rule**. It is also associated with "**2013holding**" and "**2006holding**", which would help the model process them as "**holding**".
8. "**also**": The neighbors "**see**", "**additionally**", "**<DATE4>**" and "**<NUM3>**" can point out that the high frequency of this string in citation sentences is captured. For instance, the following citation sentence from a *BVA decision* can support this claim: "**see also** Quartuccio v. Principi, 16 Vet. App. 183 (2002).".
9. "**Decision**": The fact that it has the closest distance to the strings like "**appeal**", "**unappealed**", "**veteransappeal**", "**board**", "**petition**", "**issue**", which can be interpreted as how **appeals** and **decisions** are **issued** by the **board**.
10. "**claim**": Many sentences in the BVA decisions talk about "**claim files**" and "**claim folders**," and the model was able to find the connection because two of the closest neighbors are "**file**" and "**folder**.". On the other hand, of course, there are cases where **claims** are rejected and sometimes **reopened**, and the strings "**reopen**" and "**reopened**" are successfully placed among the *10 closest neighbors*.
11. "**evidence**": As expected, nearest neighbors such as "**medical**" and "**records**" are an easy way to say that the model is very powerful, since in *BVA decisions* the use of **medical evidence** and **records** of **evidence** is fairly common. On the other hand, as the **evidence** must be **established** or **submitted**, some of the neighbors are "**must**", "**establish**" and "**submit**".

Training & Optimizing Classifiers

The first part of the classification step is to implement a featurizer that creates the feature vector that will be fed into the classifier. For this experimental project, two different featurization techniques are used. The first one is **TFIDF Featurization**, which first *fits* the *annotated texts in the training set* into a **TFIDF Vectorizer**, which is then used to *transform the annotated texts in the training, dev and test sets* into *feature vectors*. The shape of this *feature vector* is **(3019,)**. This is not the entire *feature vector* that is used, as there are two more features. One of the two additional features is **Normalized Positions**, which stores the normalized position of the sentence in the text. This feature is very useful because certain types tend to occur in the same positions, and since it is normalized, the difference between document lengths will not cause a problem. The second additional feature is **Normalized Token Count** that stores the number of tokens in the sentence, normalized by subtracting the mean and dividing by the standard deviation of the whole training set. It is important that only the training set is used to calculate those statistics as doing otherwise would give out fake test signals to the model as normally the statistics for unseen data is not known. Adding the two additional features, the shape of this whole *feature vector* becomes **(3021,)**. That is, given **N** training samples, the shape of the input and the labels would be **(N, 3021)** and **(N,)** respectively. Note that this vectorization process is applied to t*raining, dev and test sets*.

In the second featurization technique, **Word Embedding Featurization**, the word embeddings model trained is utilized. With this featurization, the aim is to use the learned

similarities to achieve higher scores as those embeddings are projections onto a continuous word space that preserves the semantic similarities between words [4]. As a bonus, it also reduces the dimension of the *feature vector* by a factor of **30**, which would make the training process faster as time is one of the most important metrics in machine learning tasks. In this featurization, the two additional features, namely **Normalized Positions** and **Normalized Token Count**, are again used and only the *feature vector* generated by **TFIDF Vectorizer** is replaced. Here, instead of creating a *feature vector* from the sentences, the embeddings vectors for the tokens are used by taking the average of the embeddings vectors for the tokens in each sentence. Note that the dimension of this *feature vector* should be the same as the dimension embeddings model, which is **(100,)**. In the end, with the additional two features, the shape of the whole *feature vector* for each sentence becomes **(102,)**. This time, given **N** training samples, the shape of the input and output of the network would be **(N, 102)** and **(N,)**.

For the classification part, **Linear Support Vector Machine Classifier** is used as the linear model and **Random Forest Classifier** is used as the non-linear model. For the hyper-parameter choice, a couple of different combinations are tried for the non-linear model. The model is run with different combinations using a widely-used hyper-parameter function, ScikitLearn's RandomizedSearchCV. As **the number of trees**, the search space was *25, 50, 75, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1250, 1500 and 1750* whereas for **the maximum depth in the trees**, the search space was *10, 15, 20, 25, 30, 35 and 40*. These parameters are used together with the **bootstrap** parameter for finding the best hyperparameters for two featurization techniques. Even if the best parameters found had the highest validation values, they will cause the classifier to overfit the training data. That is, the classifier memorizes the training data and is likely to perform worse on unseen data due to poor generalization. Since the test set remains untouched, this cannot be said with certainty at this time. However, the overfitting becomes visible by an accuracy of 100% and should be avoided to achieve a higher variance and a better generalization in unseen data. For this very reason, we will **use fewer trees in the forest** and **keep the maximum depth of the trees lower**. Keep in mind that deeper trees for simpler models result in a lot of "**pure**" branches, which causes overfitting in **Decision Trees** [5]. However, this is only true for **Word Embedding Featurization**, since its dimension is much smaller than that of **TFIDF Vectorizer**. Therefore, the best parameters for **TFIDF Vectorizer** should be kept as it really helped to make the model more complex for **TFIDF Vectorizer** with the parameters found after hyper-parameter tuning. The best performing models will be determined by observing the overfitting behavior on the *train* set and searching for the highest possible *F1* score on the *dev* test. After testing the performance on the *dev* test, the non-linear model performed better than the linear model with both featurizations. However, due to the reasons discussed above, different hyper-parameters used for both models. While the **TFIDF-based** model is trained with **800 trees** and a maximum of **35 levels** in depth, the **Embedding-based** model is trained with **100 trees** and a maximum of **15 levels** in depth. *Table 4* below shows the results on *training*, *development* and *test* sets.

Model	F1 score on Training accuracy | macro avg	F1 score on Dev accuracy | macro avg	F1 score on Test accuracy | macro avg
TFIDF-based	96% | 96%	81% | 71%	84% | 72%
Embedding-based	96% | 93%	83% | 73%	86% | 75%

Table 4: Classification Report Metrics (F1 Scores) for the best TFIDF-based and Embedding-based models

Test Set Evaluation and Comparison between Types

As can be seen from the *Table 4* above or *Figures 6 and 7* below (*test set*), both models performed similarly, with the **Embedding-based** model having slightly higher test set accuracy. For both models, almost all types were classified correctly with a really high score.

However, there are three types that were misclassified more than the others. First of all, it is easy to see false classifications for **EvidenceBasedOrIntermediateFinding** and **EvidenceBasedReasoning**, and the models have lower scores for these types. For the latter type, the situation is even more chaotic as it was mostly classified as **Evidence**, which means it has a very low recall. The results of the annotations workshop really demonstrated that it was even difficult for us two distinguish these two types from **Evidence**. Fortunately, around these types is where the models can be compared. For instance, if looked closely at the number of misclassified **EvidenceBasedOrIntermediateFinding**, the **TFIDF-based** model was less successful than the **Embedded-based** model to distinguish that type from **Evidence**. That is, the **Embedded-based** model has a higher precision for **Evidence** than the **TFIDF-based** model. On the other hand, the **Embedded-based model** had a lot of *False Positives* for type **EvidenceBasedOrIntermediateFinding** while the **TFIDF-based** model did better for this classification, achieving a visibly higher precision for the type **EvidenceBasedOrIntermediateFinding.** Additionally, the type **LegalRule** is also misclassified as **Evidence**, but not as frequent as the other two types. One would say it is a tie and the models perform similarly, and the precision and recall values are all mixed up between the models. That is the *F1* score can come in handy, and the model with the higher accuracy is chosen to be used as “the best model”, which is the **Embedding-based** model.

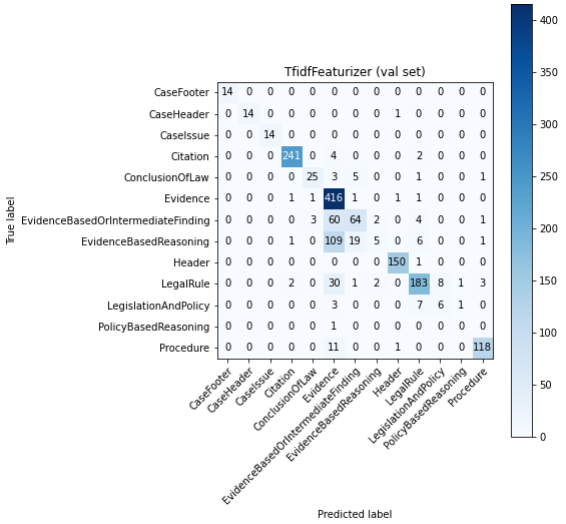

Figure 4: CM for TDIF-based model (dev set)

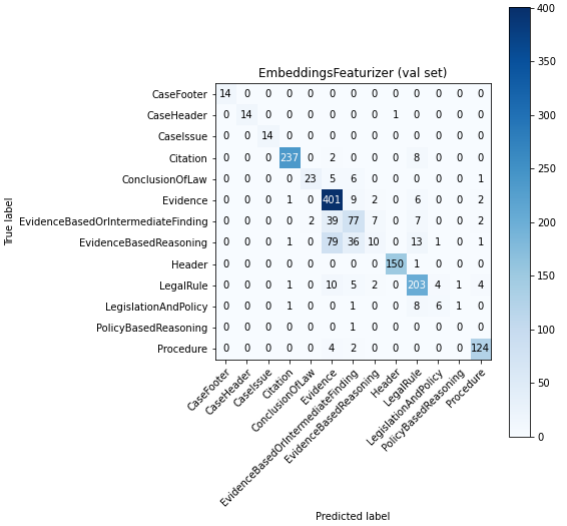

Figure 5: CM for Embeddings-based model (dev set)

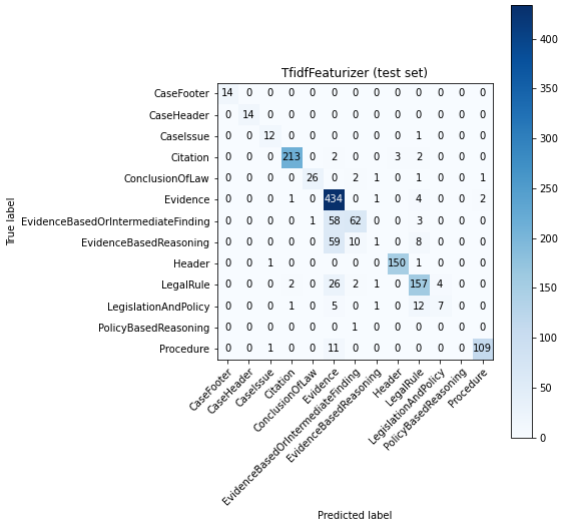

Figure 6: CM for TDIF-based model (test set)

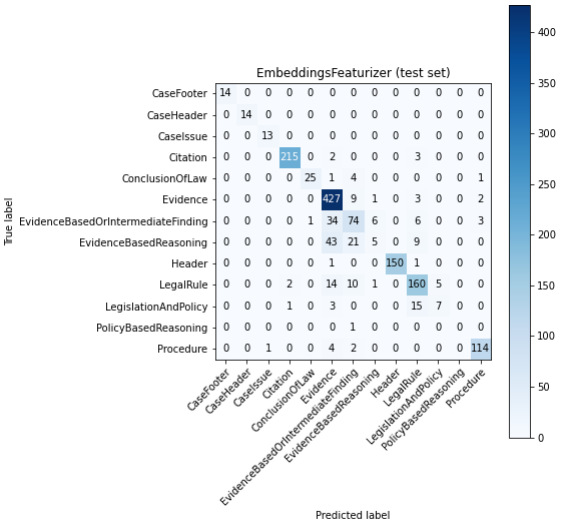

Figure 7: CM for Embeddings-based model (test set)

Note that the models were optimized several times using the scores on the dev set. Only at the end, the test set was used to check the performance of the models and to perform the final error analysis. From this point on, the model was not further optimized.

Error Analysis

As shown in *Figures 4 and 5* (*dev set*) above and discussed in the previous section, the same misclassification pattern is observed for both sets. Now that the error analysis was performed for both models, in this section, a random sample of *False Positives* across the dataset can be analyzed using the **Embedding-based** model, the decided "*best model*".

The first example is a rather simple example that without the surrounding sentences, would be tough to be classified as **EvidenceBasedReasoning**, which is the correct class. I really think that this is the feature that the model is missing, since the training results for this type are not that bad. However, currently the model classifies the following sentence as **Evidence**: "*The examiner noted that surgical menopause results from the bilateral oophorectomy.*". Another good example showing a misclassification is the following: "*The notes show that the Veteran had a lifetime history of substance abuse, clashes with authority, and poor coping.*" This sentence was **EvidenceBasedOrIntermediateFinding** but classified as **Evidence**. Investigating the sentence closely, one can see that "**Veteran had**", "**history**", "**substance abuse**", and maybe "**show**", could be the reason why the model classified this sentence as **Evidence**. Let's analyze another sentence, where the model again misclassifies **EvidenceBasedOrIntermediateFinding** as **Evidence**. The following sentence probably gets classified as **Evidence** as maybe "*he being a prisoner of war*" gave a wrong impression to the model: "*The veteran's dental problems are not as a result of trauma in service, nor was he a prisoner of war.*". As a last example, let's take this sentence: "*However, in light of the Board's grant of service connection for PTSD (by this decision), a remand to attempt to schedule the Veteran for another VA examination would merely cause avoidable delay and would not result in any additional benefit to the Veteran.*". Here, the model predicted the label as **EvidenceBasedOrIntermediateFinding** when the true label was **LegalRule**. This looks like a wrong annotation as it clearly talks about how the board's choice would cause a delay. Moreover, it is found that this attempt to schedule would cause the delay.

The second part of this section also analyzes the types that the models classified particularly well. For instance, let's look at the following sentence where the model predicted the following sentence as **Evidence**, when the true label was **Procedure**: "*The November 2000 VA cardiologist found 'no clinical evidence of valvular heart disease,' and also noted as follows:*". One can reason that the sentence contains some words such as "**evidence**", "**found**", and "**disease**", which mostly occur in **Evidence** sentences, and also a date, which is combined into a very long token, **<DATE13>**. Another example could be the following sentence annotated as **PolicyBasedReasoning**: "*In considering the evidence of record under the laws and regulations as set forth above, the Board concludes that there is no basis for compensation for any dental trauma.*" **EvidenceBasedOrIntermediateFinding** was the prediction of the model based solely on the "*Board concludes*" section, which usually follows finding of the Board. The last example is **RemandInstructions**, where it is almost impossible for the model to make a correct classification because the corpus contains a few sentences annotated as **RemandInstructions**. For example, the following sentence is classified as **Evidence**, although it is actually **RemandInstructions**: "*However, because service connection is granted by this decision for residuals of a stroke, the RO should reconsider this matter and both claims are referred to the RO for initial consideration.*". The model did a really good job here classifying this sentence as **Evidence** because the structure really looks like **Evidence** through the usual split-sentences like "*service connection is granted*" and "*claims are forwarded*".

Discussion & Lessons Learned

The BVA decision search engine developed in this experiment showed promising performance in processing decisions and inferring case outcomes. However, the overall performance in recognizing certain types of legal texts is still insufficient. The pipeline followed during the experiment provided some insights into what might be causing these problems and how the performance of the model can be improved in some cases. The experiment began like any other ML project with a balanced split of the data. The data set was split so that each set contained an equal number of each outcome, namely "approved" and "rejected". However, the problem was that once the dataset was split, the data was balanced in terms of outcomes, and since the documents were randomly split into training, development, and test datasets, there was no guarantee that the data was also balanced in terms of types, since the annotation types were not evenly distributed in the split documents. For example, the split of the data for the RemandInstructions type, although there are some, might not include that particular type and if a development set does not contain that type, it is impossible for the model to actually learn to identify it.

After splitting the dataset, the next step was to test different sentence segmenters and find out which one is best suited for which task. After a detailed segmentation analysis, it was found that while Savelka's law-specific segmenter performed best in sentence segmentation of complete decisions, for tokenizing sentences from already sentence-segmented decisions, the improved Spacy was the better choice as it produced more splits. As described in the sections above, tokens were selected that were worth going through some filters. This detailed preprocessing step was really crucial for a good classification performance. Any error would trigger a chain reaction that would affect the final results by leading to a worse embedding model. After checking the sentence tokenization process a few times, it was decided to be "*good enough*", and the tokens generated from the sentences with at least 5 tokens were fed into FastText training, which ran for **20** epochs and created a vocabulary of size **27,734**, with each word having its own *100-dimensional embedding vector*. When we manually analyzed the custom embeddings, we were confident that the representation of the embeddings was good enough, as the neighborhood analysis produced interpretable results. Of the two featurization techniques, namely TFIDF featurization and Word Embedding featurization, the overall performance of the latter was slightly better. While classification of most types was successful, the model still had problems in identifying 3 types, one of which was **EvidenceBasedReasoning**, which was mostly misclassified.

The first improvement that can be attempted as future work is to split the dataset into types rather than outcomes, since the goal of this project is to classify types rather than outcomes. In addition, errors are always possible with manual annotations. An additional sanity check of the annotations by an expert would help the whole process a lot. Also, more data could be collected so that the model can learn the rarely occurring types. Not only these, but also the feature vector could be populated with more features. For example, the high misclassification rate of **EvidenceBasedReasoning** could be fixed by adding a feature that contains the information about neighboring sentences, as during the annotation workshop, I was personally checking the neighboring sentences to decide if the type was **EvidenceBasedReasoning**. Moreover, one could also employ ensemble methods, where some models would be responsible for classifying the types, as in this project, and others could work in correctly predicting whether the type is **EvidenceBasedReasoning** or not [6].

The requirements for the project were very clearly stated and the pipeline in the project instructions could be used in many NLP projects. I never found it difficult to understand what was required of me. However, the lack of technical tools for legal classification was a bit of a challenge for me. Perhaps the project could consist of several repetitive assignments where we could discuss and compare our findings with other students.

Code Instructions

There are three entry points into this application, namely **train.py**, **train_detailed.ipynb** and **analyze.py**. While the main goal of the first two is to train classifier(s), **analyze.py** script aims to classify sentences in a given *BVA decision* using the embeddings models and the best classifier trained that saved in the *"out"* directory by running either of the train files. After setting up the local environment (*Python 3.6*) by installing the necessary packages in *requirements.txt*, one can either run **train.py** or **train_detailed.ipynb** to train a model or the ***analyze.py*** script to classify a *BVA decision*. Note that the **train.py** script contains only the minimal steps to train a classifier, while the **train_detailed.ipynb** notebook contains a more detailed pipeline where one can train and compare multiple classifiers, and debug each of the processes mentioned in the sections above. For that reason, please refer to the **train_detailed.ipynb** notebook for verifying all these results demonstrated above. The instructions for running the scripts can be found in the **main** functions of the scripts. The **analyze.py** script can be executed like ***$ python analyze.py ./data/bva_decision.txt***.

Additionally, one can run the **train.py** script from scratch without having to go through all of the time consuming processes. In order to do so, the path to the *JSON* file for the annotated documents and directory containing the unlabeled documents should be passed into the **train.py** script. In order to generate and train everything from scratch, please set **debug**, **generate_new** and **train_new** arguments to **True** while calling respective functions in the **train** function. If those are set to **False** and you have the files containing sentence segmented decisions and generated tokens in the unlabeled directory, you can run the whole process under around 2-3 minutes and can train the models from scratch. The files should be named **_sentence_segmented_decisions.json** and **_generated_tokens.json** respectively. Please note again that everything needed in this project task (*the results of each of the error analysis, the training of the different classifiers, etc.*) is included in the **train_detailed.ipynb** notebook, and the **train.py** script is only the simplified version. The project was developed following the best practices in Python so that it can be maintainable.

References

[1] Savelka, Jaromir, Vern R. Walker, Matthias Grabmair, and Kevin D. Ashley. "Sentence boundary detection in adjudicatory decisions in the united states." *Traitement automatique des langues* 58 (2017): 21.

[2] Medium. 2022. *Tokenization for Natural Language Processing*. [online] Available at: <https://towardsdatascience.com/tokenization-for-natural-language-processing-a179a891bad4#:~:text=Tokenization%20breaks%20the%20raw%20text,the%20sequence%20of%20the%20words.> [Accessed 2 March 2022].

[3] Spacy.io. 2022. [online] Available at: <https://spacy.io/> [Accessed 26 February 2022].

[4] Sahar Ghannay, Benoit Favre, Yannick Estève, and Nathalie Camelin. 2016. Word Embedding Evaluation and Combination. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 300–305, Portorož, Slovenia. European Language Resources Association (ELRA).

[5] Witten, Ian H. et al., ed. by. Data Mining: Practical Machine Learning Tools And Techniques. 3rd ed., 2011, pp. Section: 4.3, 6.1.

[6] Dietterich T.G. (2000) Ensemble Methods in Machine Learning. In: Multiple Classifier Systems. MCS 2000. Lecture Notes in Computer Science, vol 1857. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45014-9_1