Third Party Litigation Finance

Developing a framework for pricing and creating Litigation Derivatives

Johanan Anton Pranesh & Gregory Radu Colonescu

December 2024

Many thanks to Giovanni Sanchez for their help on the initial NLP.

1 Abstract

Litigation finance, which originated in Australia and later expanded to the United States, has evolved into a critical tool for both plaintiffs and defendants in modern legal practice. Initially gaining prominence during the aftermath of the Global Financial Crisis, litigation finance emerged as a strategic response to mitigate credit risk and diversify litigation portfolios. By providing capital to cover the often substantial costs of legal proceedings, third-party financiers enable litigants to pursue claims they might otherwise forgo due to financial constraints. This financing mechanism has become an essential asset for law firms, offering both a means of protecting clients' financial interests and a way to manage risk in high-stakes litigation. As litigation finance continues to grow globally, particularly in commercial disputes, class actions, and mass torts, its regulatory and ethical implications have garnered increasing attention. This paper discusses the subset of Third Party Litigation Financing and it's potential development into a tradable derivative.

Third-Party Litigation Finance (TPLF) refers to a specific form of litigation funding where an external, non-litigant investor provides capital to a party involved in a legal dispute, typically covering the costs of litigation such as legal fees, expert witnesses, and court expenses, in exchange for a portion of any financial recovery from the case's outcome. Unlike general litigation finance, which may also include funding for operational purposes or pre-litigation costs, TPLF is primarily concerned with supporting ongoing litigation in exchange for a contingent financial return.

This study aims to create a standardized contract and pricing mechanism for said contract. The pricing will be done through a neural network trained on a novel database assembled from scratch through a combination of manual data entry and scraping PDFs off the web and parsing through them using an NLP. The contract will assume the amount invested by the third party as a % of the litigation expenses will correlate 1:1 with the amount received from the law firm's share of the winnings. This would mean a 1% investment into the litigation expenses would payout 1% from the law firm's share of the winnings when all is said and done.

This is a long term project for us, and we plan on keeping you in the loop!

2 Introduction

The growing field of **third-party litigation finance** (TPLF), particularly in the context of class action lawsuits, is transforming the landscape of legal funding. Over the past two decades, third-party investors have increasingly become integral participants in financing lawsuits, offering capital to plaintiffs in exchange for a portion of any settlement or judgment.

This shift has democratized access to justice, particularly for parties with limited financial resources, and has created new opportunities for investors seeking alternative assets. One area that remains underexplored and difficult to navigate, however, is the **pricing** of third-party litigation finance products tied to class action cases, a segment of the market known for its complexity, uncertainty, and high financial stakes.

Class action lawsuits, which involve a large group of claimants collectively pursuing legal action against a common defendant, have become a critical tool for plaintiffs facing systemic legal issues. These lawsuits typically involve significant legal resources and expertise, as they deal with complex claims often spanning multiple jurisdictions and raising questions of broader social and economic importance. The financial risks for plaintiffs in these cases can be prohibitive, which is where third-party litigation finance comes in. By offering financial support for legal expenses in exchange for a percentage of any recovery, third-party funders enable plaintiffs to pursue their claims without bearing the full cost burden. This model has proven particularly effective in jurisdictions like the United States, the United Kingdom, and Australia, where litigation costs can be prohibitively high, and access to justice is often limited to those with substantial financial resources.

Despite the increasing role of third-party litigation finance, there is a notable lack of sophisticated tools for accurately pricing third-party litigation finance products related to class action lawsuits. In the absence of reliable models, investors and legal practitioners often resort to subjective assessments or heuristics based on prior experience, making pricing inconsistent and prone to error. This is particularly problematic given the need for precise valuations to determine the appropriate investment and risk-sharing arrangements between funders and plaintiffs. To address this gap, it is crucial to develop a more systematic, data-driven approach to the pricing of these financial products.

2.a Context and Motivation

The role of third-party litigation finance has grown exponentially in recent years, driven by increasing demands for capital in large-scale legal disputes and by the opportunities for high returns that these cases offer. In jurisdictions such as the United States, the UK, and Australia, third-party funders have entered the legal market, offering funding to plaintiffs in exchange for a share of the eventual award or settlement. These arrangements have not only made litigation more accessible but have also facilitated the resolution of many cases that otherwise might not have been pursued. The funding model enables plaintiffs to proceed with complex class action lawsuits, which typically require significant upfront legal costs, without the financial burden of personal investment.

Investors, too, have benefited from the emergence of third-party litigation finance, as these investments offer opportunities for potentially high returns that are uncorrelated with traditional financial markets. For investors, class action lawsuits present a high-risk, high-reward opportunity, where the value of their investment depends heavily on the success of the lawsuit. However, the complexity and uncertainty of these cases make it difficult to estimate the potential return accurately. This inherent uncertainty poses a significant challenge, as funders need to evaluate the likelihood of success, case duration, potential settlement value, and many other factors before committing substantial capital to these lawsuits.

Despite the benefits of third-party litigation finance, the pricing models available today are rudimentary and fail to incorporate the full range of factors that determine the success or failure of a case. Class action lawsuits, in particular, present unique challenges due to their complexity, large scale, and the involvement of numerous stakeholders. Each case involves a unique set of facts, jurisdictions, and legal precedents, all of which must be taken into account when determining the price of a financial product tied to the lawsuit's outcome.

The challenge lies in how to accurately assess the potential returns in the face of such uncertainty, where the variables involved are often qualitative and difficult to quantify.

There is a pressing need to develop a comprehensive pricing model that can provide reliable predictions of the potential value of third-party litigation finance products related to class action lawsuits. Such a model would not only improve transparency in the market but would also help to streamline decision-making for both investors and plaintiffs. By incorporating a wide array of strong signals, such as jurisdiction, case complexity, legal precedents, and historical case data, this model would provide more accurate, data-driven estimates of case outcomes, enabling investors to make better-informed decisions.

2.b Research Gap

One of the primary challenges in the field of third-party litigation finance, particularly with respect to class action lawsuits, is the difficulty in accurately pricing these financial products. Third-party litigation finance products share characteristics with **call derivatives**, where the value of the investment is contingent on the success of the underlying litigation. Much like a call option in traditional finance, the investor's payoff in third-party litigation finance is asymmetric—highly leveraged in the event of a successful case, but yielding little to no return if the case fails. This **option-like** structure, however, complicates pricing, as the value depends not only on the eventual outcome of the lawsuit (success or failure) but also on a range of other factors, such as case duration, jurisdictional nuances, the legal team's expertise, and the defendant's financial health. These elements introduce significant uncertainty, making third-party litigation finance products inherently complex and more difficult to model than conventional financial assets.

The challenge of pricing becomes particularly pronounced when dealing with class action lawsuits, where the number of plaintiffs, the scope of claims, and the sheer complexity of legal arguments further complicate the pricing process. Investors must navigate the uncertainty of multiple claimants, each with varying levels of involvement, and assess how factors such as judicial temperament, case strategy, and the likelihood of settlement impact the final outcome. These factors are often qualitative in nature, difficult to capture in structured data formats, and highly jurisdiction-dependent. Moreover, litigation risks are not fixed and can evolve throughout the course of the case, with unexpected rulings or new evidence potentially shifting the trajectory of the lawsuit.

Another significant gap in the literature is the lack of established frameworks for extracting and identifying meaningful features from unstructured legal data. Legal documents, such as court rulings, case filings, and expert testimonies, are typically dense with information that is not readily quantifiable or structured for machine learning algorithms. This makes it challenging to identify which variables most strongly influence the outcome of a class action lawsuit, and how these variables should be incorporated into a pricing model. Legal language is often imprecise, and the context in which a case is tried—such as jurisdictional norms, the experience of the legal team, or political considerations—can add layers of complexity that are difficult to standardize.

While machine learning and Natural Language Processing (NLP) hold significant promise for automating feature extraction from legal texts, there remains a lack of comprehensive models that can fully capture the nuances of legal language and interpret complex case details in a way that is useful for pricing third-party litigation finance products. Many existing models in the litigation finance space rely on simple heuristics or historical case data, but these methods fail to account for the evolving and dynamic nature of legal disputes. As a result, the pricing models available today are often inaccurate, inconsistent, and unable to account for the full spectrum of risk involved in third-party litigation finance.

In summary, the research gap in third-party litigation finance lies not only in the complexity of pricing models that can handle **option-like characteristics** (similar to call derivatives) but also in the **lack of standardized features** to process and analyze **unstructured legal data**. The absence of reliable features for class action lawsuits, combined with the dynamic, evolving nature of legal proceedings, presents a significant challenge to creating accurate pricing models. This gap highlights the need for new methodologies, particularly leveraging **machine learning** and **NLP**, to extract meaningful, actionable data from legal texts and integrate these insights into robust predictive models.

2.c Contributions

This paper aims to make several key contributions to the field of third-party litigation finance, focusing specifically on the pricing of class action lawsuits:

1. **Development of a Comprehensive Pricing Model**: This research introduces a **data-driven pricing model** for third-party litigation finance products associated with class action lawsuits, incorporating a broad range of **strong signals**, such as jurisdiction, case complexity, legal precedents, and other key factors. These signals are integrated into a predictive framework that allows for a more objective and scalable assessment of class action cases.

2. **Utilization of Natural Language Processing (NLP)**: One of the novel aspects of this paper is the application of **NLP techniques** to extract relevant features from unstructured legal text. By analyzing case filings, judicial opinions, and other legal documents, this model captures critical insights that inform the likelihood of success, potential settlement amounts, and case trends, which are essential for accurate pricing.

3. **Machine Learning Model for Predicting Third-Party Litigation Finance Prices**: Leveraging a **neural network-based machine learning model**, this paper develops a tool capable of predicting the price of third-party litigation finance products tied to class actions. By training the model on historical case data and strong signals identified through NLP, the tool generates more accurate forecasts of case outcomes and funding requirements.

4. **Retraining and Model Adaptation**: To account for the evolving nature of litigation and the dynamic risks associated with class action lawsuits, the model is designed to be **retrainable**. This ensures that the model can adapt to changing legal environments, market conditions, and new data, improving its accuracy over time and allowing it to remain relevant for future cases.

5. **Tradable Securities**: Finally, this paper explores the potential for **transforming predicted prices into tradable securities**, opening up opportunities for investors to engage with third-party litigation finance products in a manner akin to financial markets. By using predicted case prices as the foundation for tradable instruments, the model supports the creation of a **liquid secondary market** for third-party litigation finance, providing greater flexibility and transparency for investors.

By addressing these gaps and providing a more structured, data-driven approach to pricing, this paper lays the groundwork for more accurate, efficient, and scalable models in the third-party litigation finance industry. In doing so, it has the potential to reshape the landscape of **litigation funding**, enabling more informed decision-making, improved risk management, and broader access to capital for plaintiffs pursuing class action lawsuits.

3 Literature Review and Background

In this section, we review the existing literature on third-party litigation finance (TPLF), the methods used to price litigation finance products, and the role of machine learning models in financial forecasting. Additionally, we explore the specific application of neural networks in this field and the key innovations introduced by our proposed model. This review aims to set the stage for understanding the challenges in pricing class action lawsuits, while positioning machine learning and neural networks as a powerful solution to these challenges.

3.a Existing Methods

Third-party litigation finance is a relatively new industry, and as such, the literature on it is still evolving. While TPLF has gained prominence in legal markets, especially in jurisdictions like the United States, the United Kingdom, and Australia, comprehensive models for predicting the financial viability of these investments are sparse. Most existing approaches for pricing TPLF products are rudimentary and heavily reliant on qualitative judgment, historical case analysis, and basic financial models.

Historically, pricing litigation finance products, especially in the context of class action lawsuits, has been based on *heuristics* and expert opinion. Investors rely on experienced professionals who make subjective assessments of the likelihood of case success, the potential for settlement, and the possible duration of litigation. These qualitative evaluations often focus on factors such as the reputation of the legal team, the complexity of the case, the presence of strong precedents, and the defendant's financial condition. However, these methods lack precision and cannot fully account for the large number of variables involved in class action cases, such as evolving legal arguments, procedural delays, or changing political or economic factors that may influence the case outcome.

The application of probabilistic models and *Monte Carlo simulations* has also been explored in literature. These methods aim to account for the various uncertainties in class action cases by simulating numerous possible outcomes based on a variety of assumptions. Monte Carlo simulations can generate a range of probable outcomes, thus providing a distribution of values that investors can use to assess potential returns. While these models are more sophisticated than basic heuristics, they still face significant challenges when it comes to dealing with the complexity of unstructured legal data. In particular, extracting the right features from legal documents to inform the simulations remains a significant barrier.

Furthermore, legal analytics platforms have emerged in recent years, using historical data to generate insights into the likelihood of success in litigation. These platforms rely heavily on *statistical analysis* and *regression models* to find correlations between historical case outcomes and various factors, such as the type of case, jurisdiction, and the experience of the legal team. While these tools can provide useful benchmarks, they tend to rely on aggregated data and are limited in their ability to account for the subtleties and nuances of individual cases, especially in the context of class action lawsuits, where each case has its own set of intricacies.

Despite these advancements, the field of third-party litigation finance pricing remains underdeveloped, with existing methods often lacking the granularity, flexibility, and accuracy required for effective decision-making. Moreover, most existing methods do not leverage the full potential of modern computational techniques that can process large amounts of data, such as natural language processing (NLP) and machine learning (ML). These shortcomings create a significant opportunity for innovation, particularly through the application of more advanced models like neural networks.

3.b Why Neural Networks?

Neural networks, a subset of machine learning algorithms, have emerged as a powerful tool in solving complex problems where traditional methods have limitations. The ability of neural networks to automatically learn patterns from large and high-dimensional datasets makes them especially suitable for applications in fields like finance, where data is often unstructured and voluminous. Neural networks can be used to model non-linear relationships and interactions between variables, making them ideal for predicting complex outcomes such as those found in third-party litigation finance.

The case for using neural networks in TPLF pricing lies in their capacity to handle both structured and unstructured data. Structured data in TPLF models may include variables such as case type, jurisdiction, defendant's financial status, and historical settlement amounts. However, legal cases also produce vast amounts of unstructured data, such as text in legal filings, judicial opinions, and news reports. Extracting meaningful insights from these sources is a significant challenge, as traditional data analysis techniques are often inadequate for understanding the subtleties of legal language. Neural networks, particularly those using Natural Language Processing (NLP) techniques, are well-suited to this task, as they can analyze large quantities of textual data to uncover hidden patterns that may not be obvious to human analysts.

The versatility of neural networks also makes them well-suited to the dynamic nature of litigation. Unlike static models, which assume that the relationships between input variables remain constant, neural networks can adapt over time as they learn from new data. This ability to *retrain* the model continuously ensures that it remains relevant in an ever-changing legal and financial environment. As new court rulings are issued, new cases are filed, and emerging legal precedents come into play, a neural network-based model can adjust its predictions to account for these changes, making it far more adaptable than traditional models.

One key advantage of using neural networks for TPLF pricing is their ability to capture complex, non-linear relationships. In traditional financial models, many assumptions are made about the linearity of relationships between variables. However, in litigation finance, the interplay between various factors such as jurisdiction, legal strategy, case complexity, and the potential for settlement is highly non-linear. Neural networks, by design, excel at modeling such non-linearities, enabling them to better reflect the real-world dynamics of class action lawsuits. This is particularly important for class actions, where a combination of factors—including the scale of the case, the number of plaintiffs, and the likelihood of legal precedents—can influence the outcome in ways that are difficult to predict using conventional models.

Another critical advantage is the ability of neural networks to handle large datasets. Class action lawsuits often involve extensive documentation, including pleadings, motions, depositions, and court rulings. These documents contain a wealth of information that can provide insight into the likely outcome of a case, but processing them manually is time-consuming and prone to error. Neural networks, with their capacity for automated feature extraction from legal texts, can analyze this data more efficiently, allowing for the integration of more diverse and detailed inputs into the pricing model.

Given these advantages, the use of neural networks presents an exciting opportunity to improve the accuracy and scalability of litigation finance pricing models. By incorporating a wider range of data and adjusting to evolving case dynamics, neural networks can provide investors and legal teams with more accurate and actionable predictions. The ability to process both structured and unstructured data, adapt to new information, and capture non-linear relationships makes neural networks a natural fit for the complexities inherent in third-party litigation finance.

3.c Key Innovations

This paper introduces several key innovations that aim to address the current limitations in pricing third-party litigation finance, particularly in the context of class action lawsuits. The first innovation lies in the development of a **hybrid pricing model** that incorporates both structured and unstructured data. Traditional pricing models in litigation finance often rely solely on structured financial data and historical case outcomes. However, by integrating unstructured data, such as legal filings, court opinions, and news reports, our model is able to capture a broader range of signals that are highly relevant to predicting the outcome of class action lawsuits.

A key aspect of this innovation is the application of **Natural Language Processing (NLP)** to extract features from legal texts. NLP techniques enable the model to identify patterns in legal language, such as trends in judicial decisions, the influence of certain legal arguments, and the potential impact of case-specific factors like jurisdictional bias or recent legal precedents. By incorporating these textual insights, the model provides a much richer and more comprehensive picture of a case, leading to more accurate pricing predictions.

The second innovation is the use of a **neural network-based machine learning model** to make predictions about the potential outcomes of class action lawsuits. By leveraging deep learning techniques, the model can analyze vast amounts of historical data and identify complex patterns that are difficult to detect with traditional methods. This allows the model to generate more accurate predictions regarding the likely success or failure of a case, as well as the potential settlement value.

Finally, this paper proposes the concept of **retraining and model adaptation**. As litigation cases evolve and new data becomes available, the model can be retrained to improve its predictions. This ensures that the model remains up-to-date and relevant, capturing changes in legal trends, judicial behavior, and other factors that influence the outcome of class action lawsuits.

By addressing these challenges, the innovations presented in this paper provide a powerful tool for investors, legal teams, and other stakeholders in the third-party litigation finance market. The combination of neural networks, NLP, and retraining capabilities enables the development of a more dynamic, scalable, and accurate pricing model for class action lawsuits, improving decision-making and risk management in this rapidly growing field.

4 Data and Methodology

4.a Pre-data

The first part to get our data is first finding signals. Public Access to Court Electronic Records (PACER), the online federal court case locator, has a limit of 100 cases per quarter. Luckily, classaction.org/database has many class actions to pick from, with filters on states and the law the case was filed under, oftentimes coming with notes on who won. This allowed us to find cases with a variety of jurisdictions and legal frameworks, allowing us to see if the signal we thought we found was something that was common across all class actions or just that specific one.

The following are the signals we chose and why we chose it.

- The jurisdiction/state: Certain courts have better expertise and have ruled in favor of a particular party, for example when creditors filed a suit against a borrower in London, they filed in the southern court as it shown favorability to creditors. Sometimes, a judge or court could be driven through ideology or some other factor. Alternatively,

the court might have rules/traditions on how many witnesses can testify. This could bias the jury one way or another.

- The subject of the lawsuit (the underlying product or service) along with the probabilities that it was a product or service based off of keywords in the suit. The subject of the lawsuit is incredibly important, especially if it's something that constantly gets sued over. For example, bank fees and data breaches are a good chunk of class actions, and they win tidy sums for both the plaintiffs and law firms. A subject that has repeated wins or losses is a great signal. The probabilities are important to give the neural network a sense of how sure the NLP was in classifying the subject. A less sure prediction might mean the case wasn't well constructed and is more likely to lose.

- Probability classification if the subject is a core part of the company or an accessory (for example, an Epipen is a core part of Mylan while marketing emails for Palantir wouldn't be): We think that some companies are more willing to settle early if the subject is an accessory even if the money demanded is high and if it's a core and the money demanded is low. Early settlements could mean less money for the plaintiffs but a higher IRR, while late settlements or going to court could mean losing money. Some companies are completely the opposite: more willing to settle their main profit generator early and drag out other suits for significantly longer. So this sort of thing would be important when paired with who the defendant is.

- Average plaintiff costs: The amount of money lost by each plaintiff may affect the jury, especially if there are punitive awards. Or if it's something like a database breach due to poor cybersecurity, it may end up with no money being awarded to the plaintiffs but the law firm getting money. Regardless, this will affect the final winnings of the case.

- Whether or not it's a Delaware corporation: corporations that are located in Delaware have certain protections. We believe this would reduce the expected payoff, especially if the class action is in Delaware, as these protections could eliminate parts of the initial filing as irrelevant.

- Table of Contents: Could be an indicator of the "professionalism" of the law firm. For example, there are many small lawsuits (order of thousands or tens of thousands of dollars) with a no-name law firm that are settled for almost nothing or are dismissed in New York and are relatively unprofessional compared to the multimillion dollar lawsuits that involve thousands or tens of thousands of people with a big name law firm behind it.

- List of laws filed under: The law that the lawsuit was filed under is very important as it changes the entire nature of the lawsuit. Different laws will have different probabilities of winning, which will effect the expected payoff.

- Keywords: depending on what keywords are present there may be something important to the neural net. This for us is something more experimental- we have no idea if this will be of any importance to the neural net, but perhaps it finds some pattern.

- Magazines/Reviews: Things that show up in the media may bias the jury or be used by the plaintiffs to show their cause is worthy. Typically, lawsuits with a lot of media attention will also be led by big law firms ready to splurge in order to win, increasing the expected payout.

- Federal agencies: The US government being involved with the company before on a similar issue could indicate a culture of rulebending or a lot of evidence the plaintiffs can use to win, which we think can increase the expected value of the lawsuit.

Now that we had an idea of what we were looking for, we were able to have a game plan. We would try to look for a database with at least a portion of what we were looking for, especially the winnings, the amount the law firm made, and finally the expenses incurred by the defense. The other pieces can be scraped from the lawsuit filings.

It was time to gather our pre-data. There is no rigorous centralized database that contains all the data we need- in fact, nothing exists that contains most of the type of signals we were looking for, and Wharton Research Data Services' Federal Justice Center database had a couple of the basic indicators we were looking for like jurisdiction, but out of the three important indicators we were looking for it had none working correctly. To verify this later, we cross referenced our list of downloaded suits to the database and found that all the cases were showing as $0 or $230MM in winnings (and the vast majority were $0, which didn't line up with what we could tell). This also left us in the dark about the amount of money the law firm made and the expenses incurred, so we ended up creating a webscraper to download every pdf we could from classaction.org/database. A limitation here is that only 1,000 results show up across all the pages, so we had to build the scraper such that it would iterate through every single state and law that was visible on the page (Exhibit 1). We were lucky we did this early since building it took several hours, testing several more, and finally running it took 30+ hours (due to all the waits in the code to let the webpages load).

4.b Data Gathering

Now it was signal extracting time. Giovanni built an initial NLP that we built off of to extract most of our signals. This base file took in the pdfs, converted them to text files, and compared them against a dictionary. Then it works finds the relevant words and phrases (for example, "New York Times" when checking for magazines). We mostly refined it by adding more checks and changed file download directories, along with some other changes. We also have a classifier started by him which uses BART Large MNLI for classifying the subject between product and service. To find the jurisdiction and laws, we used the path that the lawsuit was found in (so for example, if we found it on classaction.com/database under NC > RICO > Sherman Antitrust Act, the state was North Carolina and the list of laws was [RICO, Sherman Antitrust Act]

So we have the X part of our dataset, now we need the Y. Unfortunately this had to be done manually. Luckily, consumer-action.org and topclassactions.com had a lot of cases that we were able to cross reference with what we had downloaded. This meant going through these two websites, finding the links to the class action notice websites and reviewing them in order to find all the data necessary. We were then able to manually input the winnings and the amount the law firm made (we ended up tabling the expenses for later because we weren't able to find enough data). Almost all were 1/3, .3, or .35 of the winnings, but a couple we weren't sure (so we put the industry standard of 1/3). Unfortunately because this step took so long we don't have a lot of data yet.

Exhibit 2 is a sample of our winning lawsuits.

From here it was fairly straightforward: combine our data (manual inputs, file location data, classifications, and NLP results), transform any data we needed to (for example, dictionaries of keyword frequencies would be turned into new columns), and encode any text data. Then we got to building our neural network.

4.c The Neural Network

Because our initial dataset was fairly small, we can't have anything too grand. We built it with Keras, with an input layer of 64 neurons and a ReLU activation function. Then we had a dropout layer for regularization, followed by another layer with 32 neurons and another ReLU, and finally an output layer. We used Adam as our optimizer with MSE as our loss function, while having 50 epochs and a batch size of 4 to help with our small dataset.

Some things to keep in mind is that since we are planning on building this out going forward, we still want to keep things like computational efficiency. So we used the ReLU, which is computationally efficient and it avoids the vanishing gradient problem, and applies to what we're doing (some NLP). Similarly, we used Adam because of it's computational efficiency when it comes to sparse data, ability to work with nonconvex optimization problems, ability to do bias correction, and an adaptive learning rate which helps with a dataset like ours. It works well for computer vision and with unstructured data (such as this one).

Funnily enough, the payoff of this contract we're emulating is the same as a ReLU function.

This is different from our presentation because we did more research regarding smaller datasets and figured that we needed less neurons in our model.

5 Results

5.a Test Loss and MAE

We had a standardized test loss of .001711, surprisingly low, while our MAE was .036042. This implies that our model performs significantly better when it works with standardized features and a little bit of luck in regards to our test loss. In the future we will also do cross validation in order to see the differences.

5.b Actual Results

	Perry v. Progressive	**Henry V. Brown University**
AmountWon	61,000,000	284,000,000
AmtWonPred	25,172,878	249,685,440
Diff0	35,827,122	34,314,560
Diff0%0	142	13
Diff0%1	58	12
FirmAmountWon	15,000,000	94,666,670
FirmAmtWonPred	17,919,924	72,757,600
Diff1	2,919,924	21,909,066
Diff1%0	19	23
Diff1%1	16	30

Note: Differences and percentages are absolute values.

The results here are promising but needs far more testing and data before being able to say it's rigorous.

6 Discussion

6.a Important missing signals

We're missing some pretty key signals that we hope to be able to include a later date. First would be various date indicators (year filed, month filed, etc). This would be followed up by number of named plaintiffs, the geographic spread of the plaintiffs, total damages, any plaintiff deaths, along with several details that could be taken from the civil cover sheet and many many more that couldn't be easily done with NLP.

6.b Signal Analysis

We'd like to do marginal analysis as well, as in remove one signal and see how well the model performs without that signal, and repeat. This will help us isolate important and unimportant signals, or possibly even detrimental signals.

7 Conclusion

Despite not having a large dataset, we ended up being able to predict within the same order of magnitude for our two test cases, which is not a terrible start to this long project. We believe if we had stuck it out with the entire team we could have had more data for the neural network, so team 1's project would have been doable, but team 2 would have had a difficult time creating the probability score for the model.

Our basic model is $c = f * E[\text{Payoff}] * e^{-rt}$. f here is the fraction of the initial contract cost c that the third party is willing to fund. r is the expected risk free rate, in this example assumed to be 3% and t is the expected amount of time, which is about 2.5 years. So for our two examples, a hedge fund could put in \$16.6MM for Perry v. Progressive, or \$67.5MM for Henry v. Brown University. However, this not only drastically overshoots the initial amount invested, but also doesn't account for the inherent riskiness of lawsuits. This is where team 2's analysis would have been helpful. The alternative here is that the risk-free return isn't important- instead the internal rate of return, or IRR, is. The IRR would fluctuate depending on time but the idea would be that hedge funds can 10x their investment (assuming a 10% investment to payoff ratio on wins) as their payoff would be correlated with the winnings. This way, the basic model becomes $c = f * E[\text{Payoff}]$, while the graph of the payoff becomes the same as a call with a strike price of \$0. This is because the payoff will be correlated with the final winnings of the case, while losing nothing if the payoff is negative (for example, a countersuit is filed and the plaintiffs lose). Sometimes law firms limit their earnings based off of winnings- this creates the same payoff as a call bull spread if the contract is based off of law firm winnings.

Returning to our example, a hedge fund putting in \$100,000 into either case would expect \$1,000,000 to be their winnings (10x), with some adjustment needed due to time value of money. In reality, their winnings would be \$837,057.12 for Perry v. Progressive and \$1,301,124.14 for Henry v. Brown University.

8 Exhibits

Exhibit 1- classaction.org/database options

Exhibit 2- Database sample of winnings

	Defendant	AmountWon	FirmAmountWon	FirmExpenses	state	list_of_laws	reason	product	service	core part of the company	accessory part of the company
childress-et-al-v-jp-morgan-chase-and-co-et-al.pdf	JP Morgan Chase & Co., et al.	62461938	18738581.4	NaN	North Carolina	[Servicemembers Civil Relief Act]	proprietary program	0.8902	0.1098	0.8567	0.1433
checchia-v-bank-of-america-na.pdf	Bank of America, N.A.	8000000	2666666.666667	NaN	Pennsylvania	[]	retail banking services	0.7795	0.2205	0.6716	0.3284
salls-et-al-v-digital-federal-credit-union.pdf	Digital Federal Credit Union	1800000	600000.0	NaN	Massachusetts	[]	DFCU	0.5906	0.4094	0.6167	0.3833
stevens-et-al-v-zappos-com.pdf	Zappos.com	0	1600000.0	1600000.00	Massachusetts	[]	shoes and apparel	0.9915	0.0085	0.9366	0.0634
owens-et-al-v-bank-of-america-na-et-al.pdf	Bank of America, N.A.	4950000	1650000.0	18829.07	Florida	[]	Bank of America	0.7043	0.2957	0.8053	0.1947
dasher-v-rbc-bank.pdf	RBC Bank	7500000	2625000.0	NaN	Florida	[]	retail banking services	0.7795	0.2205	0.6716	0.3284
disposable-contact-lens-antitrust-class-action.pdf	AAB Optical Group et al.	118000000	39333333.333333	1710000.00	Florida	[Sherman Antitrust Act]	disposable contact lenses	0.8820	0.1180	0.9289	0.0711
hernandez-v-wells-fargo-bank-na.pdf	Wells Fargo Bank, NA	18500000	4525000.0	335000.00	California	[]	Proprietary Software	0.9279	0.0721	0.8632	0.1368
figueroa-v-capital-one-na-et-al.pdf	Capital One	13000000	4333333.333333	NaN	California	[]	ATM Fee	0.7936	0.2064	0.6022	0.3978
walters-v-target.pdf	Target	5000000	2110900.28	NaN	California	[]	Target Debit Card	0.8672	0.1328	0.6952	0.3048
rivera-v-wells-fargo-bank.pdf	Wells Fargo	3800000	1266666.666667	NaN	California	[]	Wells Fargo	0.5190	0.4810	0.8107	0.1893
stewart-v-early-warning-services-llc.pdf	Early Warning Services, LLC.	4000000	1333333.333333	NaN	New Jersey	[Fair Credit Reporting Act]	Internal Fraud Prevention Service	0.9824	0.0176	0.7038	0.2962
henry-et-al-v-brown-university-et-al.pdf	Brown University, et al.	284000000	94666666.666667	NaN	Illinois	[Sherman Antitrust Act]	Enrollment Data Visualization Tool	0.9300	0.0700	0.6823	0.3177
gott-v-mylan-nv-et-al.pdf	Mylan et al	609000000	203000000.0	NaN	Kansas	[RICO]	EpiPen	0.9611	0.0389	0.5025	0.4975
perry-et-al-v-progressive-michigan-insurance-company-et-al.pdf	Progressive Michigan Insurance Company, et al.	61000000	15000000.0	460000.00	Michigan	[]	Progressive	0.7302	0.2698	0.5302	0.4698
hardy-v-transamerica-life-insurance-company.pdf	Transamerica Life Insurance Company	88000000	29333333.333333	NaN	Alabama	[]	General Services Life Insurance Company	0.6581	0.3419	0.7035	0.2965

AvgAmtPlaintiffCost	keywords	Delaware corporation	Table of Contents	Magazines/Reviews	Federal Agencies
4.561035e+05	{'correspondence': 1, 'payment': 1, 'reduce': ...	1	0	{'NA': 0}	{'federal': 3}
3.500000e+01	{'get': 1, 'payment': 1, 'agreement': 1, 'take...	0	0	{'NA': 0}	{'federal': 1}
5.040231e+05	{'payment': 1, 'agreement': 1, 'take': 1, 'red...	0	0	{'NA': 0}	{'attorney general': 1, 'ftc': 1, 'federal': 13}
1.250002e+06	{'payment': 1, 'contract': 1, 'take': 1, 'frau...	1	0	{'consumer reports': 6}	{'federal trade commission': 2, 'ftc': 12, 'fe...
1.859986e+04	{'get': 1, 'correspondence': 1, 'cut': 1, 'pay...	0	1	{'new york times': 5, 'consumer reports': 2}	{'sec': 1, 'federal': 51, 'fdic': 17}
4.400000e+01	{'payment': 1, 'agreement': 1, 'sign': 1, 'tak...	0	0	{'NA': 0}	{'federal': 2, 'fdic': 4}
4.007800e+01	{'get': 1, 'cut': 1, 'payment': 1, 'agreement'...	1	1	{'NA': 0}	{'federal trade commission': 2, 'ftc': 21, 'fe...
3.582696e+05	{'payment': 1, 'agreement': 1, 'contract': 1, ...	0	0	{'new york times': 4, 'forbes': 1}	{'department of justice': 1, 'federal trade co...
1.886364e+01	{'get': 1, 'payment': 1, 'agreement': 1, 'take...	0	0	{'NA': 0}	{'federal': 9, 'fdic': 10}
3.454562e+05	{'get': 1, 'payment': 1, 'agreement': 1, 'sign...	0	0	{'NA': 0}	{'federal': 4}
1.420267e+06	{'correspondence': 1, 'cut': 1, 'payment': 1, ...	0	1	{'NA': 0}	{'commissioner': 6, 'federal': 15}
0.000000e+00	{'fraud': 1, 'declaration': 1}	0	0	{'NA': 0}	{'federal': 1}
4.547543e+04	{'undertake': 1, 'agreement': 1, 'sign': 1, 'p...	0	0	{'the atlantic': 1, 'forbes': 2}	{'department of justice': 2, 'doj': 1, 'federa...
5.000345e+05	{'sham': 1, 'cut': 1, 'agreement': 1, 'exclude...	0	0	{'bloomberg': 2}	{'department of justice': 1, 'federal': 3, 'me...
3.039302e+03	{'narrow': 1, 'declaration': 1, 'payment': 1, ...	0	0	{'NA': 0}	{'federal': 2}
1.850002e+06	{'balance': 1, 'contract': 1}	0	0	{'NA': 0}	{'NA': 0}

Exhibit 3- Webscraping code

```
import os
from selenium import webdriver
from selenium.webdriver.chrome.service import Service as ChromeService
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.keys import Keys
from webdriver_manager.chrome import ChromeDriverManager
import pandas as pd
from bs4 import BeautifulSoup
import time
import pickle as pkl


download_directory = os.path.join(os.getcwd(), 'downloads')


if not os.path.exists(download_directory):
    os.makedirs(download_directory)


pickle_file = 'prev_suits.pkl'
if os.path.exists(pickle_file):
    with open(pickle_file, 'rb') as f:
        prev_suits = pkl.load(f)
else:
    prev_suits = {}

# Set up Selenium with the specified download directory
options = webdriver.ChromeOptions()
# options.add_argument('--headless') # Run in headless mode
prefs = {
    'download.default_directory': download_directory,
    'download.prompt_for_download': False,
    'download.directory_upgrade': True,
    'safebrowsing.enabled': True,
    'plugins.always_open_pdf_externally': True
}
options.add_experimental_option('prefs', prefs)

driver = webdriver.Chrome(service=ChromeService(ChromeDriverManager().install()),
    options=options)

def download_pdf(states, laws, pdf_url):
    state_list = '+'.join(map(str, states))
    c_download_directory = os.path.join(os.getcwd(), 'downloads', state_list)


    if not os.path.exists(c_download_directory):
        os.makedirs(c_download_directory)


    for law in laws:
        c_download_directory = os.path.join(c_download_directory, law)
```

```
        if not os.path.exists(c_download_directory):
            os.makedirs(c_download_directory)


    c_options = webdriver.ChromeOptions()
    c_options.add_argument('--headless') # Run in headless mode
    c_prefs = {
        'download.default_directory': c_download_directory,
        'download.prompt_for_download': False,
        'download.directory_upgrade': True,
        'safebrowsing.enabled': True,
        'plugins.always_open_pdf_externally': True
    }

    c_options.add_experimental_option('prefs', c_prefs)
    c_driver =
        webdriver.Chrome(service=ChromeService(ChromeDriverManager().install()),
        options=c_options)

    try:
        c_driver.get(pdf_url) # Navigate to the PDF URL
        print(f'Download initiated for: {pdf_url}')


        pdf_name = pdf_url.split("/")[-1]
        pdf_path = os.path.join(c_download_directory, pdf_name)

        # Wait until the PDF file appears in the download directory
        WebDriverWait(c_driver, 30).until(lambda d: os.path.exists(pdf_path))
        print(f'Download completed for: {pdf_url}')

    except Exception as e:
        print(f'Error occurred while downloading {pdf_url}: {e}')
    finally:
        c_driver.quit()

def send_keys_one_by_one(element, text, delay=0.05):
    """Send keys one by one with a delay."""
    for char in text:
        element.send_keys(char)
        time.sleep(delay)


def scrape_pdfs(state, law):

    driver.get('https://classaction.org/database')
    time.sleep(.1)


    WebDriverWait(driver, 2).until(
        EC.presence_of_element_located((By.TAG_NAME, 'body'))
    )


    state_input = WebDriverWait(driver, 2).until(
```

```
    EC.visibility_of_element_located((By.CSS_SELECTOR, '#ca-db-search-component
        > div > div >
        div.db.cf.relative.dtc-l.w-100.w-25-l.h-100.h2.white.ph3.br.b--black-05
        > div:nth-child(2) > div > div > div > form > input'))
)

time.sleep(.4)
state_input.click()
time.sleep(.4)
state_input.clear()
time.sleep(.4)
send_keys_one_by_one(state_input, state)
time.sleep(.4)
state_input.send_keys(Keys.RETURN)
time.sleep(.4)

# Wait for the law input field to be present and visible
law_input = WebDriverWait(driver, 2).until(
    EC.visibility_of_element_located((By.CSS_SELECTOR, '#ca-db-search-component
        > div > div >
        div.db.cf.relative.dtc-l.w-100.w-25-l.h-100.h2.white.ph3.br.b--black-05
        > div:nth-child(4) > div > div > div > form > input'))
)

# Click to focus on the law input field
time.sleep(.4)
law_input.click()
time.sleep(.4)
law_input.clear()
time.sleep(.4)
send_keys_one_by_one(law_input, law)
time.sleep(.4)

# Check for "No results" message
try:
    no_results = WebDriverWait(driver, 2).until(
        EC.presence_of_element_located((By.CSS_SELECTOR,
            '.ais-RefinementList-noResults'))
    )
    print(f'No results for state: {state}, law: {law}. Moving to next state/law
        combo.')
    return # Skip to the next law

except Exception:
    # No "No results" message found, continue processing
    # print('pass')
    pass

time.sleep(.1)

law_input.send_keys(Keys.RETURN)
time.sleep(.3)

# print('hits now')
# Select the number of hits per page
try:
    hits_per_page_select = WebDriverWait(driver, 10).until(
```

```
            EC.element_to_be_clickable((By.CSS_SELECTOR, '.ais-HitsPerPage-select'))
        )
        hits_per_page_select.click() # Open the dropdown

        time.sleep(.1)

        # Wait for the specific option to be clickable
        option = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.CSS_SELECTOR,
                '.ais-HitsPerPage-option:nth-child(4)'))
        )
        option.click() # Click the option

        # Wait for results to refresh after changing the selection
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CSS_SELECTOR,
                'a.dib.br-pill.ba.b--near-white-blueish.no-underline'))
        )

    except Exception as e:
        print(f'Error occurred while selecting hits per page: {e}')

    # Iterate through pages
    while True:
        # Wait for the results to appear
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CSS_SELECTOR,
                'a.dib.br-pill.ba.b--near-white-blueish.no-underline'))
        )

        # Parse the page
        soup = BeautifulSoup(driver.page_source, 'html.parser')

        # Select PDF links
        pdf_links =
            soup.select('a.dib.br-pill.ba.b--near-white-blueish.no-underline')

        for link in pdf_links:
            pdf_url = link['href']
            if not pdf_url.lower().endswith('.pdf'):
                continue
            if not pdf_url.startswith('http'):
                pdf_url = 'https://classaction.org' + pdf_url


            if pdf_url in prev_suits.keys():
                if state not in prev_suits[pdf_url]['state']:
                    prev_suits[pdf_url]['state'].append(state)
                if law not in prev_suits[pdf_url]['law']:
                    prev_suits[pdf_url]['law'].append(law)
            else:
                prev_suits[pdf_url]= {'state':[state,], 'law':[law,]}

        # Check for the next page button
        try:
            next_page_button = WebDriverWait(driver, 10).until(
```

```
                EC.element_to_be_clickable((By.CSS_SELECTOR,
                    '#ca-db-search-component > div > div >
                    div.db.cf.relative.dtc-l.w-100.w-75-l.bg-near-white.v-top >
                    div.ais-Pagination.mb4 > ul >
                    li.ais-Pagination-item.ais-Pagination-item--nextPage > a'))
            )
            if 'disabled' in next_page_button.get_attribute('class'):
                # print("No more pages to scrape.")
                break # Exit the loop if the button is disabled
            next_page_button.click() # Click the next page button
            print('next page')
            WebDriverWait(driver, 10).until(
                EC.presence_of_element_located((By.CSS_SELECTOR,
                    '#ca-db-search-component > div > div >
                    div.db.cf.relative.dtc-l.w-100.w-75-l.bg-near-white.v-top >
                    div.ais-Pagination.mb4 > ul >
                    li.ais-Pagination-item.ais-Pagination-item--nextPage > a'))
            )
        except Exception as e:
            # print("Error occurred while clicking next page:", e)
            break # Exit the loop if there are no more pages
    print('current combo done')

if __name__ == '__main__':
    all_states = ['Alabama', 'Alaska', 'Arizona', 'Arkansas', 'California',
        'Colorado', 'Connecticuit', 'Delaware', 'District of Columbia', 'Florida',
        'Georgia', 'Hawaii', 'Idaho', 'Illinois', 'Indiana', 'Iowa', 'Kansas',
        'Kentucky', 'Louisiana', 'Maine', 'Maryland', 'Massachusetts', 'Michigan',
        'Minnesota', 'Mississippi', 'Missouri', 'Montana', 'Nebraska', 'Nevada',
        'New Hampshire', 'New Jersey', 'New Mexico','New York','North Carolina',
        'North Dakota', 'Northern Mariana Islands', 'Ohio', 'Oklahoma', 'Oregon',
        'Pennsylvania', 'Puerto Rico', 'Rhode Island', 'South Carolina', 'South
        Dakota', 'Tennessee', 'Texas', 'Utah', 'Vermont', 'Virgin Islands',
        'Virginia', 'Washington', 'West Virginia', 'Wisconsin', 'Wyoming', 'zCourt
        of Federal Claims']
    laws =
        list(pd.read_csv('Background_Scripts_and_Files/modified_laws.csv')['law_name'])

    # states = ['Alabama']
    # laws = ['Telephone Consumer Protection Act']

    done_states = []
    for i in prev_suits:
        if prev_suits[i]['state'][0] not in done_states:
            done_states.append(prev_suits[i]['state'][0])
    done_states.sort()
    skip_scraping = False
    try:
        last_state = done_states[-1]
        if last_state == "zCourt of Federal Claims":
            skip_scraping = True
        index = all_states.index(last_state)
        states = all_states[index + 1:] # Everything to the right
    except:
        states = all_states.copy()

    if not skip_scraping:
```

```
    for state in states:
        for law in laws:
            scrape_pdfs(state, law)

        # Specify the filename for the pickle file
        filename = 'prev_suits.pkl'
        # Save the dictionary to a pickle file
        with open(filename, 'wb') as file:
            pkl.dump(prev_suits, file)

for i in prev_suits.keys():
    print(i, prev_suits[i])
    download_pdf(prev_suits[i]['state'], prev_suits[i]['law'], i)
driver.quit()
print('fully done')
```

Exhibit 4- 0create_results_dict.ipynb

```
import pickle as pkl
from rapidfuzz import fuzz, process
import pandas as pd

# results_dict has format pdf_name:gain/loss
try:
    with open('results_dict.pkl', 'rb') as f:
        results_dict = pkl.load(f)
except:
    results_dict = {}

with open('Download_PDFs_py_files/prev_suits.pkl', 'rb') as f:
    prev_suits = pkl.load(f)
# Function to extract plaintiff and defendant from the URL
def extract_plaintiff_and_defendant(url):
    # Extract the part before the '.pdf' extension
    case_name = url.split('/')[-1].replace('.pdf', '')

    # Split the case name by '-v-' to separate plaintiff and defendant
    if '-v-' in case_name:
        parts = case_name.split('-v-', 1) # Split only once
        plaintiff = parts[0].strip()
        defendant = parts[1].strip() if len(parts) > 1 else ""
        plaintiff = plaintiff.replace("-", " ")
        defendant = defendant.replace("-", " ")
        # print(plaintiff)
        return plaintiff.strip(), defendant.strip()
    else:
        return None, None

prev_suits_list = []
for k in prev_suits.keys():
    prev_suits_list.append(extract_plaintiff_and_defendant(k))

def search_for_lawsuit(plaintiff, defendant, threshold = 80):
    case_and_score = []
    for case in prev_suits_list:
        plaintiff_score = fuzz.ratio(plaintiff, case[0])
        defendant_score = fuzz.ratio(defendant, case[1])
        avg_score = (plaintiff_score + defendant_score) / 2 # Combine scores
        case_and_score.append((case, plaintiff_score, defendant_score, avg_score))
    # Filter by threshold
    filtered = [r for r in case_and_score if r[3] >= 40]
    filtered = [r for r in filtered if r[2] >= 40]
    filtered = [r for r in filtered if r[1] >= 40]
    # print(plaintiff, defendant, sorted(filtered, key=lambda x: x[3],
        reverse=True))
    return sorted(filtered, key=lambda x: x[3], reverse=True)

cases = [
    'Checchia v. Bank of America, N.A.', # Downloaded separately
    'Gott v. Mylan et al',
    "Figueroa v. Capital One",
    "Hardy v. Transamerica Life Insurance Company",
```

```
    "Hernandez v. Wells Fargo Bank, NA", # 5
    "Stewart v. Early Warning Services, LLC.",
    "Walters v. Target",
    "Rivera v. Wells Fargo",
    "Unknown v. AAB Optical Group et al.",
    "Dasher v. RBC Bank", # 10
    "Owens, et al. v. Bank of America, N.A.",
    "Gary and Anne Childress, et al. v. JP Morgan Chase & Co., et al.",
    "Perry et al. v. Progressive Michigan Insurance Company, et al.",
    "Henry, et al. v. Brown University, et al.",
    "Salls et al. v. Digital Federal Credit Union",# 15
    "Stevens et al. v. Zappos.com",
]
for case in cases:
    # case = 'Checchia v. Bank of America, N.A.'
    plaintiff, defendant = case.split(' v. ', 1)
    a = search_for_lawsuit(plaintiff, defendant)

case_keys = [
    "checchia-v-bank-of-america-na.pdf",
    "",
    "figueroa-v-capital-one-na-et-al.pdf",
    "hardy-v-transamerica-life-insurance-company.pdf",
    "hernandez-v-wells-fargo-bank-na.pdf", # 5
    "stewart-v-early-warning-services-llc.pdf",
    "walters-v-target.pdf",
    "rivera-v-wells-fargo-bank.pdf",
    "disposable-contact-lens-antitrust-class-action.pdf",
    "dasher-v-rbc-bank.pdf", # 10
    "owens-et-al-v-bank-of-america-na-et-al.pdf",
    "childress-et-al-v-jp-morgan-chase-and-co-et-al.pdf",
    "perry-et-al-v-progressive-michigan-insurance-company-et-al.pdf",
    "henry-et-al-v-brown-university-et-al.pdf",
    "salls-et-al-v-digital-federal-credit-union.pdf", # 15
    "stevens-et-al-v-zappos-com.pdf",
]
unk = 1/3
case_results = [
    (8000000,  1/3,    -1),
    (264000000+345000000, 1/3, -1),
    (13000000, unk,    -1),
    (88000000, unk,    -1),
    (18500000, 4525000, 335000), # 5
    (4000000,  unk,    -1),
    (5000000,  2110900.28, -1),
    (3800000,  unk,    -1),
    (118000000, 1/3,   1710000),
    (7500000,  .35,    -1), # 10
    (4950000,  1650000, 18829.07),
    (62461938, .3,     -1),
    (61000000, 15000000, 460000),
    (284000000, 1/3, -1),
    (1800000, unk, -1), # 15
    (0, 1600000, 1600000),
]
i = 0
for case in cases:
```

```
    # print('c', case)

    plaintiff, defendant = case.split(' v. ', 1)
    # print('d', defendant)
    rez_list = search_for_lawsuit(plaintiff, defendant)
    if len(case_keys[i]) > 0:
        # print(case_keys[i])
        results_dict[case_keys[i]] = (defendant, case_results[i][0],
            case_results[i][1], case_results[i][2])
    else:
        # print('finding')
        new_key_tup = rez_list[0][0]
        # print('nkt', new_key_tup)
        new_key = new_key_tup[0] + ' v ' + new_key_tup[1] + '.pdf'
        new_key = new_key.replace(' ', '-')
        # print('nk', new_key)
        results_dict[new_key] = (defendant, case_results[i][0], case_results[i][1],
            case_results[i][2])
    i += 1
    # print()


with open('results_dict.pkl', 'wb') as f:
    pkl.dump(results_dict, f)
```

Exhibit 5- 1read_pdf.py

```
import os
from tika import parser
import pickle as pkl

class ScrapingDataToTxt:
    def __init__(self, base_directory, results_dict, output_directory):
        self.base_directory = os.path.abspath(base_directory)
        self.results_dict = results_dict
        self.output_directory = os.path.abspath(output_directory)

        # Ensure the output directory exists
        os.makedirs(self.output_directory, exist_ok=True)
        print(f"Processing directory: {self.base_directory}")
        print(f"Text files will be saved to: {self.output_directory}")

    def update_results_dict(self):
        updated_results_dict = {}

        # Stack for DFS-like traversal
        stack = [(self.base_directory, [])] # Each element is a tuple (directory,
            subdirectories list)

        while stack:
            current_dir, subdirectories = stack.pop()

            for file in os.listdir(current_dir):
                file_path = os.path.join(current_dir, file)
                if os.path.isdir(file_path):
                    # Add subdirectory to the stack and continue
                    stack.append((file_path, subdirectories + [file]))
                elif file.lower().endswith(".pdf") and file.lower() in
                    self.results_dict:
                    # Get the old tuple (Bank name, amount, etc.) and add state and
                        subdirectory list
                    old_value = self.results_dict[file.lower()]
                    try:
                        updated_results_dict[file.lower()] = old_value +
                            (subdirectories[0], subdirectories[1:])
                    except:
                        try:
                            updated_results_dict[file.lower()] = old_value +
                                (subdirectories[0], [])
                        except:
                            updated_results_dict[file.lower()] = old_value + ('', [])


        return updated_results_dict

    def pdfToTxt(self):
        count = 0
        # Walk through all directories and subdirectories
        for root, _, files in os.walk(self.base_directory):
            for file in files:
                # Check if the file is a PDF and exists in the updated results_dict
```

```
                if file.lower().endswith(".pdf") and file.lower() in
                    self.results_dict:
                    # print()
                    # print(file)

                    # Get the relative path of the file to preserve the directory
                        structure
                    relative_path = os.path.relpath(root, self.base_directory)
                    # print(f"Relative Path: {relative_path}")

                    # Construct the output path based on the relative path
                    # output_dir = os.path.join(self.output_directory, relative_path)
                    # os.makedirs(output_dir, exist_ok=True)

                    # Define the output text file path
                    txt_filename = f"{file.lower().replace('.pdf', '.txt')}"
                    # txt_path = os.path.join(output_dir, txt_filename) # Ensure
                        unique output path
                    txt_path = os.path.join(self.output_directory, txt_filename) #
                        Ensure unique output path

                    try:
                        print(f"Processing file: {file}...")
                        pdfFilePath = os.path.join(root, file)

                        # Parse PDF content
                        data = parser.from_file(pdfFilePath)
                        content = data.get('content', '').strip()

                        if content:
                            # Clean up and format content
                            formatted_content = " ".join(content.split())

                            # Write to the corresponding .txt file
                            with open(txt_path, "w", encoding='utf-8') as f: # Open
                                with 'w' to write fresh content
                                f.write(f"{formatted_content}\n")
                            count += 1
                        else:
                            print(f"No content found in {file}. Skipping...")

                    except Exception as e:
                        print(f"Error processing {file}: {e}")

        print(f"Processed {count} files.")

if __name__ == '__main__':
    # Load the results_dict from the pickle file
    with open('results_dict.pkl', 'rb') as f:
        results_dict = pkl.load(f)
    for k, v in results_dict.items():
        print(k, v)
    print()

    # Initialize the class and update the results_dict
    prepData = ScrapingDataToTxt(r"./downloads", results_dict, r"./lawcases")
    results_dict = prepData.update_results_dict()
```

```
    # Save the updated results_dict back to a pickle file (optional)
    with open('results_dict.pkl', 'wb') as f:
        pkl.dump(results_dict, f)

    # Run the processing function
    prepData.pdfToTxt()
    print()
    for k, v in results_dict.items():
        print(k, v)
```

```
# import os
# from tika import parser
# import pickle as pkl

# class ScrapingDataToTxt:
#     def __init__(self, base_directory, results_dict, output_directory):
#         self.base_directory = os.path.abspath(base_directory)
#         self.results_dict = results_dict
#         self.output_directory = os.path.abspath(output_directory)

#         # Ensure the output directory exists
#         os.makedirs(self.output_directory, exist_ok=True)
#         print(f"Processing directory: {self.base_directory}")
#         print(f"Text files will be saved to: {self.output_directory}")

#     def pdfToTxt(self):
#         count = 0
#         # Walk through all directories and subdirectories
#         for root, _, files in os.walk(self.base_directory):
#             for key in self.results_dict:
#                 # Check if the key is part of the filename (case insensitive)
#                 for file in files:
#                     if file.lower().endswith(".pdf") and key in file.lower():
#                         pdfFilePath = os.path.join(root, file)
#                         txt_filename = f"{key}.txt"
#                         txt_path = os.path.join(self.output_directory, txt_filename)

#                         try:
#                             print(f"Processing file: {file}...")
#                             # Parse PDF content
#                             data = parser.from_file(pdfFilePath)
#                             content = data.get('content', '').strip()

#                             if content:
#                                 # Clean up and format content
#                                 formatted_content = " ".join(content.split())

#                                 # Write to the corresponding .txt file
```

```
#                             with open(txt_path, "a", encoding='utf-8') as f:
#                                 f.write(f"{formatted_content}\n")
#                             count += 1
#                         else:
#                             print(f"No content found in {file}. Skipping...")

#                     except Exception as e:
#                         print(f"Error processing {file}: {e}")

#         print(f"Processed {count} files.")

# if __name__ == '__main__':
#     # Load the results_dict from the pickle file
#     with open('results_dict.pkl', 'rb') as f:
#         results_dict = pkl.load(f)

#     # Initialize the class and run the processing function
#     prepData = ScrapingDataToTxt(r"./downloads", results_dict, r"./lawcases")
#     prepData.pdfToTxt()









# import os
# from tika import parser
# import pickle as pkl

# class ScrapingDataToTxt:
#     def __init__(self, directory):
#         self.directory = os.path.abspath(directory)
#         print(f"Processing directory: {self.directory}")

#     def pdfToTxt(self):
#         print("Looping through files...")
#         count = 0
#         for file in os.listdir(self.directory):
#             if file.lower().endswith(".pdf"):
#                 print(f"Opening file {file}...")
#                 pdfFilePath = os.path.join(self.directory, file)
#                 print("Parsing...")
#                 data = parser.from_file(pdfFilePath)
#                 Data = data.get('content', '')
#                 if not Data:
#                     print(f"No content found in {file}. Skipping...")
#                     continue
#                 try:
#                     print("Writing to txt file...")
#                     strippedData = Data.strip()
#                     formatData = " ".join(strippedData.split())
#                     txt_filename = f"{file[:-4]}.txt"
```

```
#                   txt_path = os.path.join(self.directory, txt_filename)
#                   with open(txt_path, "a", encoding='utf-8') as f:
#                       f.write(f"{formatData}\n")
#                   count += 1
#               except Exception as e:
#                   print(f"Error processing {file}: {e}")
#                   continue
#       print(f"Processed {count} files.")

# if __name__ == '__main__':
#     with open('results_dict.pkl', 'rb') as f:
#         results_dict = pkl.load(f)
#     prepData = ScrapingDataToTxt(r"./lawcases")
#     prepData.pdfToTxt()
```

Exhibit 6- 2legal_case_classifier_1.0.py

```
import os
import re
import csv
import torch
from transformers import pipeline

Q1 = "This is a legal case. What is the company's product or service involved in
    this legal case?"
Q2 = ["product", "service"]
Q3 = ["core part of the company", "accessory part of the company"]

# THANK YOU GIOVANNI

def initialize_pipelines():
    # Check if CUDA (GPU) is available
    if torch.cuda.is_available():
        device = 0 # GPU device index (0 is the first GPU)
        print("GPU detected. Using GPU for inference.")
    else:
        device = -1 # CPU
        print("GPU not detected. Using CPU for inference.")

    # Initialize zero-shot classification pipeline
    classifier = pipeline(
        "zero-shot-classification",
        model="facebook/bart-large-mnli",
        device=device
    )

    # Initialize question-answering pipeline
    qa_pipeline = pipeline(
        "question-answering",
        model="deepset/roberta-base-squad2", # More performant model
        tokenizer="deepset/roberta-base-squad2",
        device=device
    )

    return classifier, qa_pipeline

def read_case_text(file_path):
    if not os.path.exists(file_path):
        print(f"Error: The file '{file_path}' does not exist.")
        return ""

    with open(file_path, 'r', encoding='utf-8') as file:
        content = file.read()

    if not content.strip():
        print(f"Error: The file '{file_path}' is empty.")
        return ""

    # Find the index after "INTRODUCTION"
    introduction_pattern = re.compile(r'\bINTRODUCTION\b', re.IGNORECASE)
    match = introduction_pattern.search(content)
```

```
    if match:
        # Extract text after "INTRODUCTION"
        start_index = match.end()
        content_after_intro = content[start_index:].strip()
        return content_after_intro
    else:
        print("Warning: 'INTRODUCTION' section not found. Using the entire text.")
        return content.strip()

def extract_product_service(text, qa_pipeline):
    # First attempt: Use QA pipeline with a specific question
    question = Q1
    try:
        result = qa_pipeline(question=question, context=text)
        answer = result.get('answer', '').strip()
        if answer and answer.lower() != "not identified":
            print(f"QA Extraction: {answer}")
            return answer
    except Exception as e:
        print(f"Error during QA extraction: {e}")

    return "Not identified"

def classify_item(text, classifier, labels):
    """
    Classifies a specific item (product or service) using zero-shot classification.

    Args:
        text (str): Text related to the item to classify.
        classifier: Zero-shot classification pipeline.
        labels (list): List of candidate labels for classification.

    Returns:
        tuple: Top label and its corresponding score.
    """
    result = classifier(text, labels, multi_label=False)
    top_label = result['labels'][0]
    top_score = result['scores'][0]
    return top_label, top_score

def display_classification_results(classifications):
    print("\n--- Classification Results ---")

    # Identified Product/Service
    print("1. Identified Product/Service:")
    print(f"\t- {classifications['identified_item']}")

    # Classification for Product or Service
    print("2. Is the case related to a product or a service?")
    for label, score in zip(classifications['product_or_service']['labels'],
        classifications['product_or_service']['scores']):
        print(f"\t- {label.capitalize()}: {score*100:.2f}%")

    # Classification for Core or Accessory
    print("3. Is this product/service a core or accessory part of the company?")
    for label, score in zip(classifications['role_in_company']['labels'],
        classifications['role_in_company']['scores']):
```

```
        print(f"\t- {label.capitalize()}: {score*100:.2f}%")
    print()

def save_results_to_csv(classifications, output_file="classification_results.csv"):

    output_dir = './csv_classification_output'
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)


    file_exists = os.path.exists(f'{output_dir}/{output_file}')


    row = [
        classifications['case name']+'.pdf',
        classifications['identified_item'], # reason (identified product/service)
        f"{classifications['product_or_service']['scores'][0]*100:.2f}%", # product
            score
        f"{classifications['product_or_service']['scores'][1]*100:.2f}%", # service
            score
        f"{classifications['role_in_company']['scores'][0]*100:.2f}%", # core part
            score
        f"{classifications['role_in_company']['scores'][1]*100:.2f}%" # accessory
            part score
    ]


    with open(f'{output_dir}/{output_file}', mode='a', newline='',
        encoding='utf-8') as csv_file:
        writer = csv.writer(csv_file)


        if not file_exists:
            writer.writerow(['case name','reason', 'product %', 'service %', 'core
                part of the company', 'accessory part of the company'])


        writer.writerow(row)

    print(f"Results saved to {output_file}")

import os

def main():
    restart_classification_result = True
    output_file = "csv_classification_output/classification_results.csv"
    if os.path.exists(output_file) and restart_classification_result:
        os.remove(output_file)
        print(f"{output_file} has been deleted.")
    else:
        print(f"{output_file} does not exist or False.")


    classifier, qa_pipeline = initialize_pipelines()

    # Dynamically get all file paths in the ./lawcases directory
    directory = "./lawcases"
```

```
file_paths = [os.path.join(directory, file) for file in os.listdir(directory)
    if file.endswith(".txt")]

# Loop through each file path
for file_path in file_paths:
    print(f"\nProcessing file: {file_path}")

    # Read the case text after "INTRODUCTION"
    input_text = read_case_text(file_path)

    if not input_text:
        print(f"Skipping file: {file_path} (No valid content found)")
        continue

    # Extract the identified product/service
    identified_item = extract_product_service(input_text, qa_pipeline)
    ajmnd = os.path.splitext(os.path.basename(file_path))[0]

    # Prepare the classifications dictionary
    classifications = {
        "case name": ajmnd,
        "identified_item": identified_item,
        "product_or_service": {},
        "role_in_company": {}
    }

    if identified_item.lower() != "not identified":
        # Classification 2: Is it a product or a service?
        labels_q2 = Q2
        label2, score2 = classify_item(identified_item, classifier, labels_q2)
        classifications["product_or_service"] = {
            "labels": [label2, "service" if label2 == "product" else "product"],
            "scores": [score2, 1 - score2]
        }

        # Classification 3: Is it a core or accessory part of the company?
        labels_q3 = Q3
        label3, score3 = classify_item(identified_item, classifier, labels_q3)
        classifications["role_in_company"] = {
            "labels": [label3, "accessory part of the company" if label3 ==
                "core part of the company" else "core part of the company"],
            "scores": [score3, 1 - score3]
        }
    else:
        # If no product/service identified, assign neutral scores
        classifications["product_or_service"] = {
            "labels": Q2,
            "scores": [-1.0, -1.0]
        }
        classifications["role_in_company"] = {
            "labels": Q3,
            "scores": [-1.0, -1.0]
        }


    display_classification_results(classifications)
```

```
        save_results_to_csv(classifications,
            output_file="classification_results.csv")

if __name__ == "__main__":
    main()



# def main():
#     # Initialize pipelines
#     classifier, qa_pipeline = initialize_pipelines()

#     # Define a list of file paths to process
#     file_paths = [
#         "./lawcases/law_case_1.txt",
#         "./lawcases/law_case_2.txt", # Add more file paths as needed
#         "./lawcases/law_case_3.txt",
#         # ...
#     ]

#     # Loop through each file path
#     for file_path in file_paths:
#         print(f"\nProcessing file: {file_path}")

#         # Read the case text after "INTRODUCTION"
#         input_text = read_case_text(file_path)

#         if not input_text:
#             print(f"Skipping file: {file_path} (No valid content found)")
#             continue

#         # Extract the identified product/service
#         identified_item = extract_product_service(input_text, qa_pipeline)
#         ajmnd = os.path.splitext(os.path.basename(file_path))[0]
# #         print(ajmnd)
#         # Prepare the classifications dictionary
#         classifications = {
#             "case name":ajmnd,
#             "identified_item": identified_item,
#             "product_or_service": {},
#             "role_in_company": {}
#         }

#         if identified_item.lower() != "not identified":
#             # Classification 2: Is it a product or a service?
#             labels_q2 = Q2
#             label2, score2 = classify_item(identified_item, classifier, labels_q2)
#             classifications["product_or_service"] = {
#                 "labels": [label2, "service" if label2 == "product" else
    "product"],
#                 "scores": [score2, 1 - score2]
#             }

#             # Classification 3: Is it a core or accessory part of the company?
```

```
#             labels_q3 = Q3
#             label3, score3 = classify_item(identified_item, classifier, labels_q3)
#             classifications["role_in_company"] = {
#                 "labels": [label3, "accessory part of the company" if label3 ==
    "core part of the company" else "core part of the company"],
#                 "scores": [score3, 1 - score3]
#             }
#         else:
#             # If no product/service identified, assign neutral scores
#             classifications["product_or_service"] = {
#                 "labels": Q2,
#                 "scores": [-1.0, -1.0]
#             }
#             classifications["role_in_company"] = {
#                 "labels": Q3,
#                 "scores": [-1.0, -1.0]
           # }
#
#         # Display the classification results
#         display_classification_results(classifications)
#
#         # Save the results to the CSV file with a fixed name
    "classification_results.csv"
#         save_results_to_csv(classifications,
    output_file="classification_results.csv")
#
# if __name__ == "__main__":
#     main()
```

Exhibit 7- 2processor_2.0.py

```
import os
import re
import nltk
import pickle
from nltk.corpus import wordnet
from nltk.tokenize import sent_tokenize


nltk.download('punkt')
nltk.download('wordnet')
nltk.download('omw-1.4')
nltk.download('stopwords')

class TextFileProcessor:
    def __init__(self, directory, keys_file, output_dir, agency_terms_file,
        magazines_file):
        self.directory = os.path.abspath(directory)
        self.keys_file = keys_file
        self.output_dir = os.path.abspath(output_dir)
        self.agency_terms_file = agency_terms_file
        self.magazines_file = magazines_file
        self.signals = {}
        self.create_output_directory()
        self.product_synonyms = set()
        self.service_synonyms = set()
        self.agency_terms = set()
        self.magazines = set()
        self.load_agency_terms()
        self.load_magazine_terms()

    def create_output_directory(self):
        if not os.path.exists(self.output_dir):
            os.makedirs(self.output_dir)
            print(f"Created output directory at {self.output_dir}")
        else:
            print(f"Output directory already exists at {self.output_dir}")

    def load_keys_and_synonyms(self):
        print("Loading keywords and their synonyms...")
        try:
            with open(self.keys_file, 'r', encoding='utf-8') as f:
                lines = [line.strip() for line in f if line.strip()]
        except FileNotFoundError:
            print(f"Keywords file not found: {self.keys_file}")
            return


        keywords = set()
        for line in lines:
            split_keywords = [word.strip().lower() for word in line.split(',') if
                word.strip()]
            keywords.update(split_keywords)
```

```
        self.keys_with_synonyms = set()
        for key in keywords:
            self.keys_with_synonyms.add(key)
            # Add synonyms from WordNet
            for syn in wordnet.synsets(key):
                for lemma in syn.lemmas():
                    synonym = lemma.name().lower().replace('_', ' ')
                    self.keys_with_synonyms.add(synonym)

        print(f"Total keywords and synonyms loaded: {len(self.keys_with_synonyms)}")

    def get_synonyms(self, word):
        synonyms = set()
        for syn in wordnet.synsets(word):
            for lemma in syn.lemmas():
                synonym = lemma.name().lower().replace('_', ' ')
                synonyms.add(synonym)
        return synonyms

    def load_product_service_synonyms(self):
        print("Loading synonyms for 'product' and 'service'...")
        self.product_synonyms = self.get_synonyms('product')
        self.service_synonyms = self.get_synonyms('service')
        print(f"Product synonyms loaded: {len(self.product_synonyms)}")
        print(f"Service synonyms loaded: {len(self.service_synonyms)}")

    def load_agency_terms(self):
        print("Loading agency terms from file...")
        try:
            with open(self.agency_terms_file, 'r', encoding='utf-8') as f:
                for line in f:
                    term = line.strip().lower()
                    if term:
                        self.agency_terms.add(term)
            print(f"Total agency terms loaded: {len(self.agency_terms)}")
        except FileNotFoundError:
            print(f"Agency terms file not found: {self.agency_terms_file}")

    def load_magazine_terms(self):
        print("Loading magazine terms from file...")
        try:
            with open(self.magazines_file, 'r', encoding='utf-8') as f:
                for line in f:
                    magazine = line.strip().lower()
                    if magazine:
                        self.magazines.add(magazine)
            print(f"Total magazines loaded: {len(self.magazines)}")
        except FileNotFoundError:
            print(f"Magazines file not found: {self.magazines_file}")

    def find_prices(self, text):
        # Enhanced regex to capture various price formats
        price_patterns = [
            r'(?:  |\$| )\s?\d{1,3}(?:,\d{3})*(?:\.\d{2})?',
            r'\d{1,3}(?:,\d{3})*\s?(?:|\$| )',
            r'\$\d+'
        ]
```

```
        prices = []
        for pattern in price_patterns:
            matches = re.findall(pattern, text)
            prices.extend(matches)
        return prices

    def find_percentages(self, text):
        # Regex to find percentages
        percentage_pattern = r'\b\d+(?:\.\d+)?\s?%\b'
        percentages = re.findall(percentage_pattern, text)
        return percentages

    def find_keys(self, sentence):
        found_keys = set()
        sentence_lower = sentence.lower()
        for key in self.keys_with_synonyms:
            # Use word boundaries to ensure whole word matching
            if re.search(r'\b' + re.escape(key) + r'\b', sentence_lower):
                found_keys.add(key)
        return found_keys

    def find_synonyms(self, sentence, synonyms_set):
        found_synonyms = []
        sentence_lower = sentence.lower()
        for synonym in synonyms_set:
            # Use word boundaries to ensure whole word matching
            matches = re.findall(r'\b' + re.escape(synonym) + r'\b', sentence_lower)
            if matches:
                found_synonyms.extend(matches)
        return found_synonyms

    def find_phrase(self, text, phrase):
        return phrase.lower() in text.lower()

    def find_agency_terms(self, text):
        found_agencies = []
        text_lower = text.lower()
        for term in self.agency_terms:
            # Use word boundaries to ensure whole word matching
            matches = re.findall(r'\b' + re.escape(term) + r'\b', text_lower)
            if matches:
                found_agencies.extend(matches)
        return found_agencies

    def find_magazines(self, text):
        found_magazines = []
        text_lower = text.lower()
        for magazine in self.magazines:
            # Use word boundaries to ensure whole word matching
            matches = re.findall(r'\b' + re.escape(magazine) + r'\b', text_lower)
            if matches:
                found_magazines.extend(matches)
        return found_magazines

    def process_files(self):
        self.load_keys_and_synonyms()
        self.load_product_service_synonyms()
```

```
print("Starting text file processing...")
for filename in os.listdir(self.directory):
    if filename.lower().endswith(".txt"):
        file_path = os.path.join(self.directory, filename)
        print(f"Processing file: {filename}...")
        try:
            with open(file_path, 'r', encoding='utf-8') as f:
                text = f.read()


            sentences = sent_tokenize(text)


            signals_in_file = {
                'prices': [],
                'price_sentences': [],
                'keys': [],
                'key_sentences': [],
                'percentages': [],
                'percentage_sentences': [],
                'product_synonyms': {},
                'service_synonyms': {},
                'delaware_corporation': False,
                'table_of_contents': False,
                'magazines': {},
                'agencies': {}
            }

            # Check for specific phrases in the whole text
            signals_in_file['delaware_corporation'] = self.find_phrase(text,
                'Delaware corporation')
            signals_in_file['table_of_contents'] = self.find_phrase(text,
                'Table of Contents')

            # Find agencies and magazines in the entire text
            agencies_found = self.find_agency_terms(text)
            for agency in agencies_found:
                signals_in_file['agencies'][agency] =
                    signals_in_file['agencies'].get(agency, 0) + 1

            magazines_found = self.find_magazines(text)
            for magazine in magazines_found:
                signals_in_file['magazines'][magazine] =
                    signals_in_file['magazines'].get(magazine, 0) + 1

            for sentence in sentences:
                # Check for prices
                prices = self.find_prices(sentence)
                if prices:
                    signals_in_file['prices'].extend(prices)
                    signals_in_file['price_sentences'].append(sentence.strip())

                # Check for keywords/synonyms
                keys = self.find_keys(sentence)
                if keys:
                    signals_in_file['keys'].extend(keys)
                    signals_in_file['key_sentences'].append(sentence.strip())
```

```
                    # Check for percentages
                    percentages = self.find_percentages(sentence)
                    if percentages:
                        signals_in_file['percentages'].extend(percentages)
                        signals_in_file['percentage_sentences'].append(sentence.strip())

                    # Check for product synonyms
                    product_syns_found = self.find_synonyms(sentence,
                        self.product_synonyms)
                    for syn in product_syns_found:
                        signals_in_file['product_synonyms'][syn] =
                            signals_in_file['product_synonyms'].get(syn, 0) + 1

                    # Check for service synonyms
                    service_syns_found = self.find_synonyms(sentence,
                        self.service_synonyms)
                    for syn in service_syns_found:
                        signals_in_file['service_synonyms'][syn] =
                            signals_in_file['service_synonyms'].get(syn, 0) + 1


                signals_in_file['keys'] = list(set(signals_in_file['keys']))

                self.signals[filename] = signals_in_file

                # Save signals in a text file
                self.save_signals_to_txt(filename, signals_in_file)

            except Exception as e:
                print(f"Error processing {filename}: {e}")
                continue
        print(f"Processed {len(self.signals)} files.")


    def save_signals_to_txt(self, filename, signals):
        innerpricelist=[]
        innerkeydict={}
        innermags={}
        inneragents={}
        base_filename = os.path.splitext(filename)[0]
        output_file = os.path.join(self.output_dir, f"{base_filename}_signals.txt")
        try:
            with open(output_file, 'w', encoding='utf-8') as f:
                f.write(f"Signals for file: {filename}\n\n")
                innerfilename=os.path.splitext(filename)[0] + ".pdf"
                # Write down prices
                if signals['prices']:
                    f.write("Prices found:\n")
                    for price, sentence in zip(signals['prices'],
                        signals['price_sentences']):
                        f.write(f"- {price} in sentence: \"{sentence}\"\n")
                        innerpricelist.append(float(price[1:].replace(',', '')))
                else:
                    f.write("No prices found.\n")
                    innerpricelist.append(0)
```

```
f.write("\n")

# Write down the keywords/synonyms
if signals['keys']:
    f.write("Keywords/Synonyms found:\n")
    for key, sentence in zip(signals['keys'],
        signals['key_sentences']):
        f.write(f"- {key} in sentence: \"{sentence}\"\n")
        innerkeydict[key]=1
else:
    f.write("No keywords or synonyms found.\n")
    innerkeydict['NA']=0

f.write("\n")

# Write down the percentages
if signals['percentages']:
    f.write("Percentages found:\n")
    for percentage, sentence in zip(signals['percentages'],
        signals['percentage_sentences']):
        f.write(f"- {percentage} in sentence: \"{sentence}\"\n")
else:
    f.write("No percentages found.\n")

f.write("\n")

# Write down product synonyms
if signals['product_synonyms']:
    f.write("Product synonyms found:\n")
    for synonym, count in signals['product_synonyms'].items():
        f.write(f"- {synonym}: {count} times\n")
else:
    f.write("No product synonyms found.\n")

f.write("\n")

# Write down service synonyms
if signals['service_synonyms']:
    f.write("Service synonyms found:\n")
    for synonym, count in signals['service_synonyms'].items():
        f.write(f"- {synonym}: {count} times\n")
else:
    f.write("No service synonyms found.\n")

f.write("\n")

# Write down if "Delaware corporation" is found
if signals['delaware_corporation']:
    f.write("\"Delaware corporation\" was found in the document.\n")
    innercorporation=1
else:
    f.write("\"Delaware corporation\" was not found in the
        document.\n")
    innercorporation=0
f.write("\n")

# Write down if "Table of Contents" is found
```

```
            if signals['table_of_contents']:
                f.write("\"Table of Contents\" was found in the document.\n")
                innertable=1
            else:
                f.write("\"Table of Contents\" was not found in the document.\n")
                innertable=0

            f.write("\n")

            # Write down magazines found
            if signals['magazines']:
                f.write("Magazines/Reviews found:\n")
                for magazine, count in signals['magazines'].items():
                    f.write(f"- {magazine}: {count} times\n")
                    innermags[magazine]=count
            else:
                f.write("No magazines or reviews found.\n")
                innermags['NA']=0

            f.write("\n")

            # Write down agencies found
            if signals['agencies']:
                f.write("Federal agencies or terms found:\n")
                for agency, count in signals['agencies'].items():
                    f.write(f"- {agency}: {count} times\n")
                    inneragents[agency]=count
            else:
                f.write("No federal agencies or terms found.\n")
                inneragents['NA']=0

            # write for new dict
            entry3_results_dict[innerfilename]=[innerpricelist,innerkeydict,innercorporation,inne


        except Exception as e:
            print(f"Error writing signals to {output_file}: {e}")

    def report_signals(self):
        report = "\nSignals found in files:"
        for filename, signals in self.signals.items():
            report += f"\n\nFile: {filename}"
            # Report Prices
            if signals['prices']:
                report += "\n Prices found:"
                for price, sentence in zip(signals['prices'],
                    signals['price_sentences']):
                    continue#report += f"\n\t- {price} in sentence: \"{sentence}\""
            else:
                report += "\n\tNo prices found."

            # Report Keywords
            if signals['keys']:
                report += "\n Keywords/Synonyms found:"
                for key, sentence in zip(signals['keys'], signals['key_sentences']):
                    continue#report += f"\n\t- {key} in sentence: \"{sentence}\""
            else:
```

```
    report += "\n\tNo keywords or synonyms found."

# Report Percentages
if signals['percentages']:
    report += "\n Percentages found:"
    for percentage, sentence in zip(signals['percentages'],
        signals['percentage_sentences']):
        continue#report += f"\n\t- {percentage} in sentence:
            \"{sentence}\""
else:
    report += "\n\tNo percentages found."

# Report Product Synonyms
if signals['product_synonyms']:
    report += "\n Product synonyms found:"
    for synonym, count in signals['product_synonyms'].items():
        report += f"\n\t- {synonym}: {count} times"
else:
    report += "\n\tNo product synonyms found."

# Report Service Synonyms
if signals['service_synonyms']:
    report += "\n Service synonyms found:"
    for synonym, count in signals['service_synonyms'].items():
        report += f"\n\t- {synonym}: {count} times"
else:
    report += "\n\tNo service synonyms found."

# Report Delaware corporation
if signals['delaware_corporation']:
    report += "\n \"Delaware corporation\" was found in the document."
else:
    report += "\n \"Delaware corporation\" was not found in the
        document."

# Report Table of Contents
if signals['table_of_contents']:
    report += "\n \"Table of Contents\" was found in the document."
else:
    report += "\n \"Table of Contents\" was not found in the document."

# Report Magazines
if signals['magazines']:
    report += "\n Magazines/Reviews found:"
    for magazine, count in signals['magazines'].items():
        report += f"\n\t- {magazine}: {count} times"
else:
    report += "\n\tNo magazines or reviews found."

# Report Agencies
if signals['agencies']:
    report += "\n Federal agencies or terms found:"
    for agency, count in signals['agencies'].items():
        report += f"\n\t- {agency}: {count} times"
else:
    report += "\n\tNo federal agencies or terms found."
```

```
        # Print the report to the console
        print(report)

        # Save the final report to a text file
        self.save_final_report(report)

    def save_final_report(self, report):
        output_file = os.path.join(self.output_dir, "~final_report.txt")
        try:
            with open(output_file, 'w', encoding='utf-8') as f:
                f.write(report)
            print(f"\nFinal report saved to {output_file}")
        except Exception as e:
            print(f"Error writing final report to {output_file}: {e}")

    def save_final_dict(self):

        innerfile_name = "entry3_results_dict.pkl"
        innerfile_path = os.path.join('./', innerfile_name)
        try:
            with open(innerfile_path, "wb") as pickle_file:
                pickle.dump(entry3_results_dict, pickle_file)
            print(f"\nFinal dict saved to {innerfile_path}")
        except Exception as e:
            print(f"Error writing final dict to {innerfile_path}: {e}")



if __name__ == '__main__':
    global entry3_results_dict
    entry3_results_dict={}
    # Define the file paths
    directory = r"./lawcases"             # Directory containing the .txt files to
        process
    keys_file = r"./signals_input/keys_sentence.txt"    # File containing the
        keywords
    agency_terms_file = r"./signals_input/agency_term.txt" # File containing
        agency terms
    magazines_file = r"./signals_input/magazines.txt"  # File containing magazine
        names
    output_dir = r"./signals_output"      # Directory where results will be saved
        (will be created if it doesn't exist)

    # Create an instance of the processor
    processor = TextFileProcessor(directory, keys_file, output_dir,
        agency_terms_file, magazines_file)

    # Process the files
    processor.process_files()

    # Generate and save the final report
    processor.report_signals()
    processor.save_final_dict()
```

Exhibit 8- 3data_processor.ipynb

```
import pandas as pd
import numpy as np
import pickle as pkl

# %run 0create_results_dict.ipynb
# Load the results_dict from the pickle file
with open('results_dict.pkl', 'rb') as f:
    results_dict = pkl.load(f)
# for k, v in results_dict.items():
#     print(k, v)

### Entry 3
with open('entry3_results_dict.pkl', 'rb') as f:
    entries_3 = pkl.load(f)

dataset = pd.DataFrame([], columns = [
    'Case',
    'Defendant', 'AmountWon', 'FirmAmountWon', 'FirmExpenses', 'state',
        'list_of_laws',# from results_dict
    'reason', 'product', 'service', 'core part of the company', 'accessory part of
        the company', # from csv
    'AvgAmtPlaintiffCost', 'keywords', 'Delaware corporation', 'Table of
        Contents', 'Magazines/Reviews', 'Federal Agencies',# from report
])
dataset.set_index('Case', inplace=True)

entries_2 = pd.read_csv('csv_classification_output/classification_results.csv',
    index_col='case name')
case_list = results_dict.keys()
for case in case_list:
    print(case)
    entry1 = list(results_dict[case])
    # entry2 = ['amogus', 0, 0, 0, 0]
    entry2 = entries_2.loc[case,:]
    entry3 = entries_3[case] # case has structure: [0, 1, 2, 3, 4, 5], {'c':9/11,
        'd':69}, 1, 0, {'c':9/11, 'd':69}, {'e':.8181, 'f':69}]
    # entry3 = [[0, 1, 2, 3, 4, 5], {'amogus':9/11, 'morbius':69}, 1, 0,
        {'c':9/11, 'd':69}, {'e':.8181, 'f':69}]
    try:
        a = entry3[0]
        # print(entry3)
        # print(a)
        entry3[0] = sum(a)/len(a)
    except:
        a = 0
    row = entry1.copy()
    row.extend(entry2)
    row.extend(entry3)
    # print('r', row)
    new_row = pd.DataFrame([row], index = [case], columns=dataset.columns)
    # print(new_row)
    dataset = pd.concat([dataset, new_row])
```

```
dataset.loc[dataset['FirmAmountWon'] < 1, 'FirmAmountWon'] =
    (dataset['FirmAmountWon'] * dataset['AmountWon'])
dataset = dataset.replace(-1, np.nan)
percentage_columns = ['product', 'service', 'core part of the company', 'accessory
    part of the company']

for col in percentage_columns:
    dataset[col] = dataset[col].replace({'%': ''}, regex=True).astype(float) / 100

# 2

# 3
dataset['AvgAmtPlaintiffCost'] = dataset['AvgAmtPlaintiffCost'].apply(np.mean)
dataset

all_terms = set()
dataset['list_of_laws'].apply(lambda x: all_terms.update(x))

for term in all_terms:
    dataset[term] = dataset['list_of_laws'].apply(lambda x: 1 if term in x else 0)
dataset.drop(columns='list_of_laws', inplace=True)

dataset_expanded = dataset['keywords'].apply(pd.Series)
dataset = pd.concat([dataset.drop(columns='keywords'), dataset_expanded], axis=1)

dataset_expanded = dataset['Magazines/Reviews'].apply(pd.Series)
dataset = pd.concat([dataset.drop(columns='Magazines/Reviews'), dataset_expanded],
    axis=1)

dataset_expanded = dataset['Federal Agencies'].apply(pd.Series)
dataset = pd.concat([dataset.drop(columns='Federal Agencies'), dataset_expanded],
    axis=1)


text_col = ['defendant', 'reason']
# numeric_col = ["AmountWon", "FirmAmountWon", "FirmExpenses", "product",
    "service", "core part of the company", "accessory part of the company",
    "AvgAmtPlaintiffCost", "Delaware corporation", "Table of Contents"]

dataset.iloc[:,4:] = dataset.iloc[:,4:].replace(np.nan, 0)
dataset.to_csv('9dataset.csv')
```

Exhibit 9- 4neural_net.ipynb

```
import pandas as pd
import numpy as np

from scipy.sparse import csr_matrix

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout


gpus = tf.config.list_physical_devices('GPU')
# if gpus:
#     print(f"GPUs found: {gpus}")
# else:
#     print("No GPUs found.")

for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)




dataset = pd.read_csv('9dataset.csv')
dataset.set_index('Unnamed: 0', inplace=True)

target_columns = ['AmountWon', 'FirmAmountWon', 'FirmExpenses']
X = dataset.drop(target_columns, axis=1)
y = dataset[target_columns]
y.drop('FirmExpenses', inplace = True, axis=1) # Not enough data for the time being


text_columns = ['Defendant', 'reason', 'state']
numerical_columns = [col for col in X.columns if col not in text_columns and col
    != 'index']


preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_columns),
        ('text', OneHotEncoder(handle_unknown='ignore'), text_columns)
    ])


pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor)
])
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1,
    random_state=r_s)


X_train_transformed = pipeline.fit_transform(X_train)
X_test_transformed = pipeline.transform(X_test)


X_train_transformed = csr_matrix(X_train_transformed).toarray()
X_test_transformed = csr_matrix(X_test_transformed).toarray()

scaler_y = MinMaxScaler()
y_train_normalized = scaler_y.fit_transform(y_train)
y_test_normalized = scaler_y.transform(y_test)


model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train_transformed.shape[1],)),
    Dropout(0.2),
    Dense(32, activation='relu'),
    Dense(y_train_normalized.shape[1], activation='linear')
])

model.compile(optimizer='adam', loss='mse', metrics=['mae'])


history = model.fit(
    X_train_transformed, y_train_normalized,
    validation_data=(X_test_transformed, y_test_normalized),
    epochs=50,
    batch_size=4,
    verbose=1
)


test_loss, test_mae = model.evaluate(X_test_transformed, y_test_normalized,
    verbose=0)
print(f"Test Loss: {test_loss}, Test MAE: {test_mae}")

predictions_normalized = model.predict(X_test_transformed)
predictions = scaler_y.inverse_transform(predictions_normalized)
print('done')

from copy import deepcopy
comparison = deepcopy(y_test)

comparison['AmtWonPred'] = predictions[:, 0]
comparison['FirmAmtWonPred'] = predictions[:, 1]
for col in comparison.columns:
    comparison.col = comparison[col].astype(int)
comparison['Diff0'] = (comparison['AmountWon'] -
    comparison['AmtWonPred']).astype(int)
comparison['Diff0%0'] =
    (comparison['Diff0']/comparison['AmtWonPred']*100).astype(int)
```

```
comparison['Diff0%1'] =
    (comparison['Diff0']/comparison['AmountWon']*100).astype(int)
comparison['Diff1'] = (comparison['FirmAmountWon'] -
    comparison['FirmAmtWonPred']).astype(int)
comparison['Diff1%0'] =
    (comparison['Diff1']/comparison['FirmAmountWon']*100).astype(int)
comparison['Diff1%1'] =
    (comparison['Diff1']/comparison['FirmAmtWonPred']*100).astype(int)
new_column_order = ['AmountWon', 'AmtWonPred', 'Diff0', 'Diff0%0', 'Diff0%1',
    'FirmAmountWon', 'FirmAmtWonPred', 'Diff1', 'Diff1%0', 'Diff1%1']
comparison = comparison[new_column_order]
comparison
```