Bug Triaging Using Natural Language Processing

An approach to classify emailed agile studio bugs to respective backlogs

28 January 2020 | Jake Epstein & Matt Kenney

PEGA®

THE PROBLEM

Bugs submitted from agile studio must currently be read and triaged to the relevant backlogs by hand, taking many man hours. This process is slow and inefficient, and scrum teams often don't receive relevant bugs until months after submission

THE APPROACH

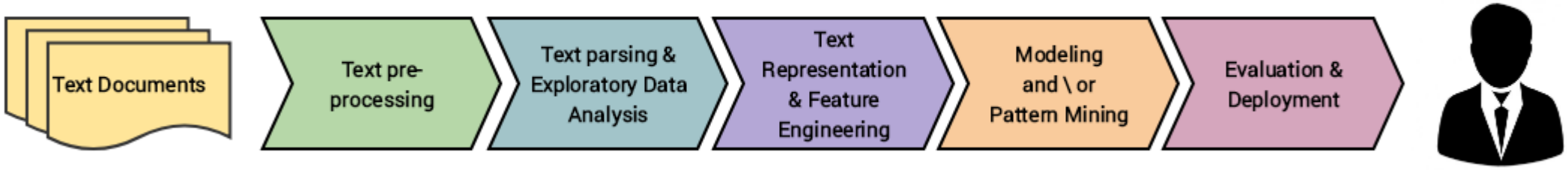
Develop a machine learning model using natural language processing and trained with historic classified bug submissions to triage bugs with high accuracy

PROBLEM CLASS

A Multi-class Text Classification Problem

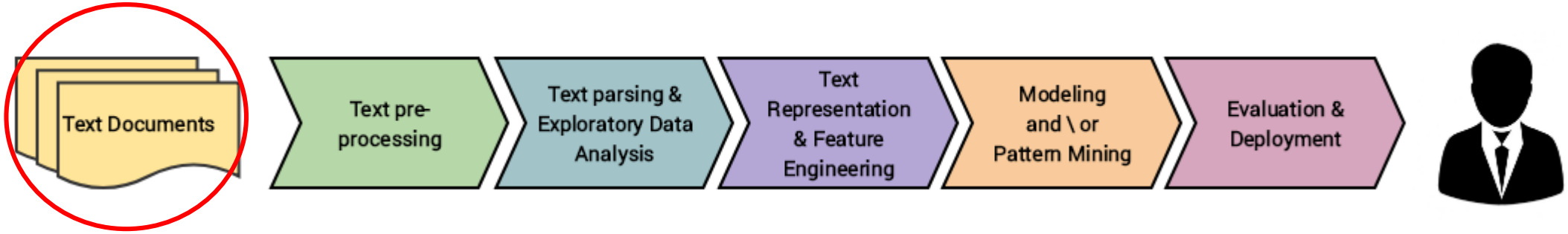
Given a series of texts and N predefined buckets, classify each text into only one of the buckets

THE NLP DEVELOPMENT METHOD



1. Start with a corpus of text containing the most inferenceable data possible
2. Follow standard text wrangling, pre-processing, and parsing
3. Based on initial insights – represent the text using relevant feature engineering techniques
4. Based on problem – focus on building predictive supervised or unsupervised model
5. Use stakeholder feedback to evaluate success criteria, deploy final model for use

THE NLP METHOD



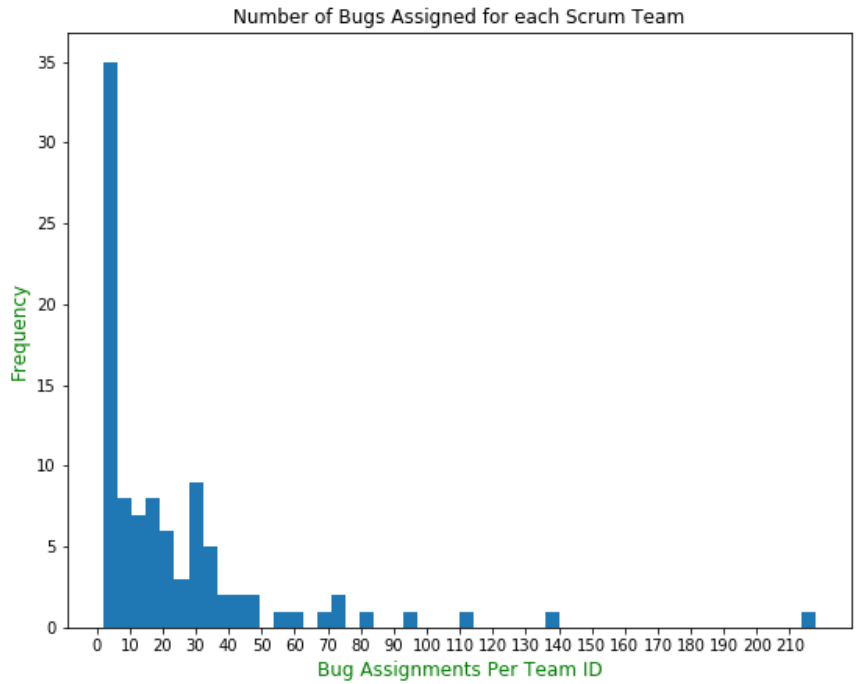
GIVEN DATASETS

The following datasets were queried from an internal Pega server and given:

1. 3 Years of Email Subject Lines:
2. 2000 Most Recent Bug Entry Subject Lines:
3. 10000 Most Recent Bug Entry Subject Lines + Description:

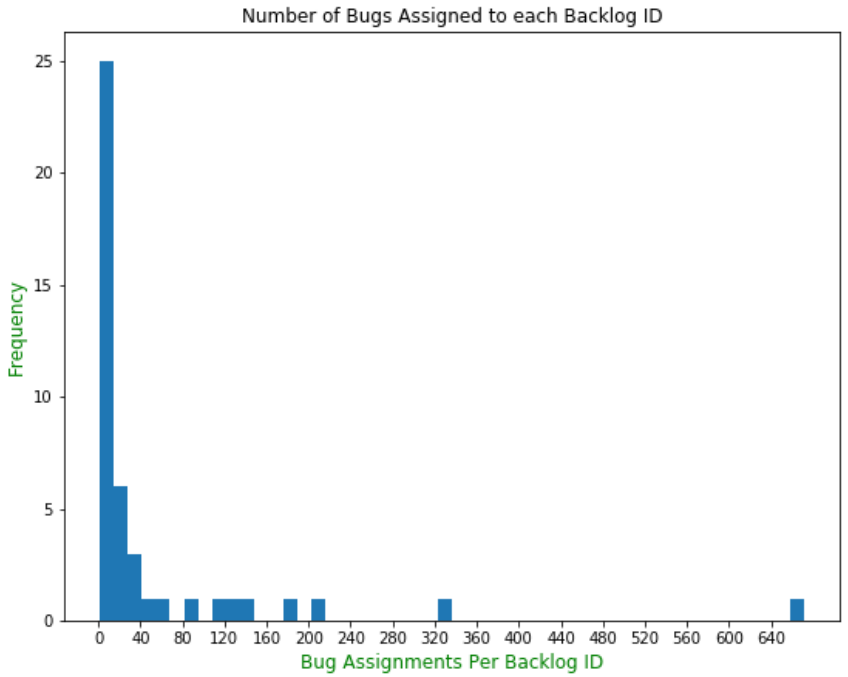
The first step: decide which data set to use

3 YRS OF EMAIL SUBJECT LINES:



Uneven distribution of training examples. Many teams were assigned fewer than 5 bugs and some as little as 1. Models overfitted the data as there were not enough training examples

3 YRS OF EMAIL SUBJECT LINES:



Bug assignments per backlog ID showed worse characteristics. Uneven distribution of training examples. Some backlog IDs had as many as 650 assignments and others as few as 1. Models still overfitted the data.

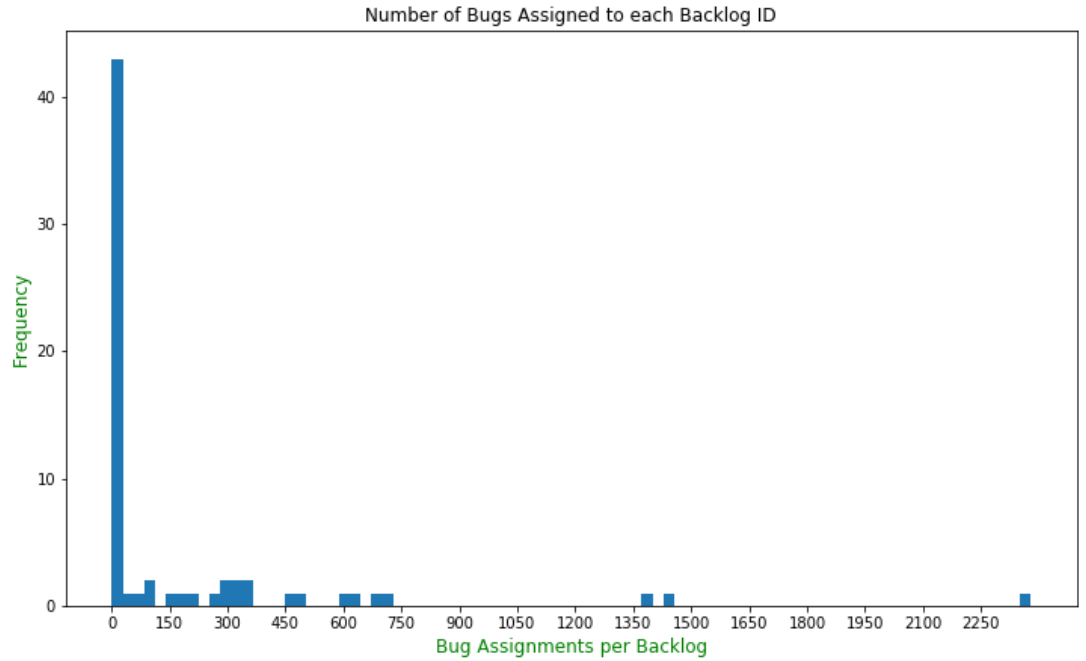
3 YRS OF EMAIL SUBJECT LINES:

Example of generic subject line with little valuable info:

"Fix Cancel Button on search landing page."

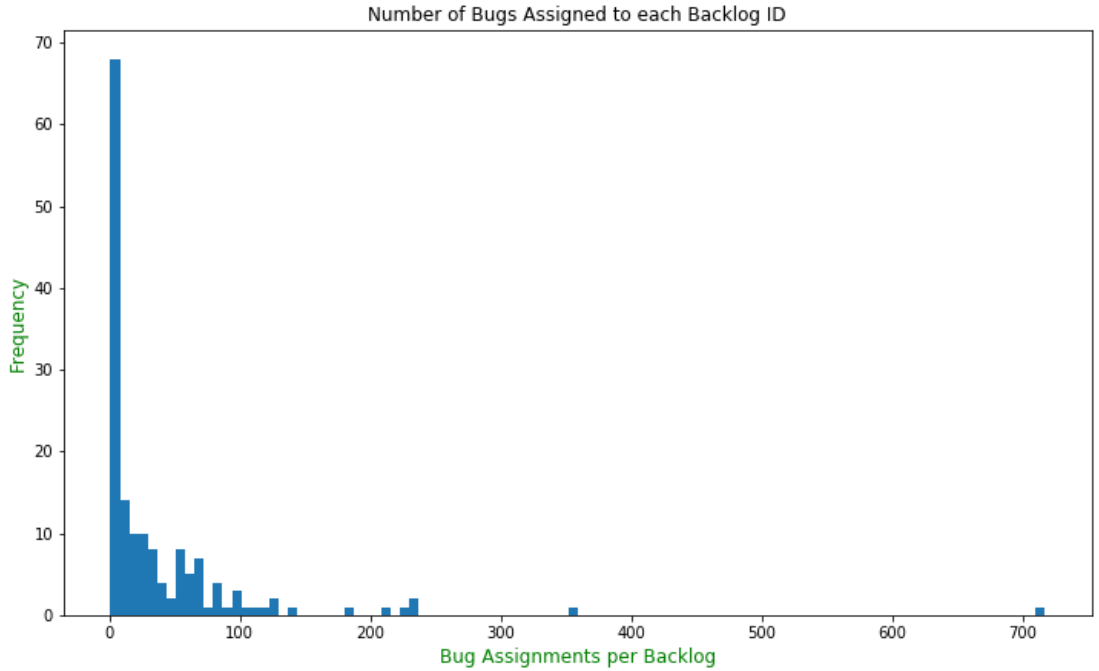
Generally, many subject lines were uninferenceable i.e. they contained too little information to be accurately classified

THE 2000 MOST RECENT BUGS FOR PLATFORM



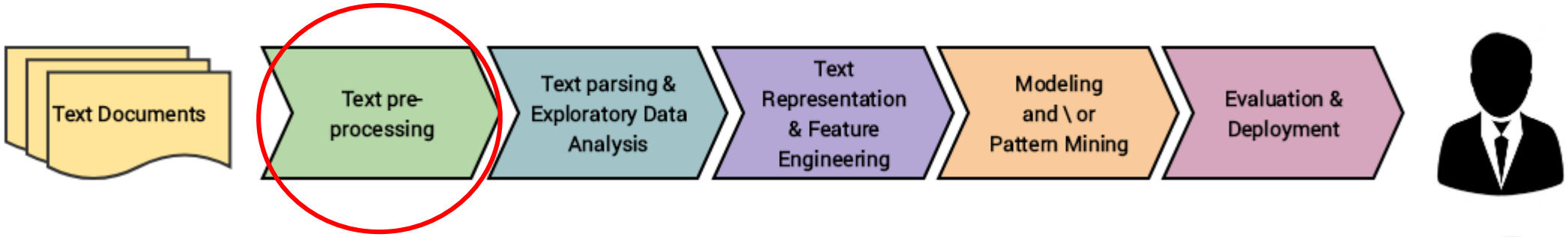
Data skewed and many labels/email subject lines still uninferenceable – containing very little identifiable information

THE 10000 MOST RECENT BUGS FOR PLATFORM WITH DESCRIPTION



Data set included labels and descriptions of bugs reported from all streams. Email bodies from emailed reports mapped to description field. Data still skewed but there is much more identifiable information useful for classification. Biased data established as a characteristic of the problem.

THE NLP METHOD



PRE-PROCESSING

Procedure

1. Remove all entries with null team ids
2. Remove all entries with null backlog ids
3. Remove all entries with null descriptions
4. Drop all entries with duplicate labels
5. Advanced cleaning methods

PRE-PROCESSING: TEXT

Accented Characters

Remove accents:
é → e

Contractions

Expand contractions:

Don't → do not

Lemmatization

Group forms of a word to be analyzed as a single item:

am, are, is → be

getting, got → get

Lowercase

Lowercase all elements:

Always → always

Punctuation

Generally remove all punctuation:

banana. → banana

Special Characters

Remove all special characters (i.e. Unicode characters)

PRE-PROCESSING: TEXT

Stop Words

Remove Stop Words – the most common words in a language:

buy a banana at market → buy banana market

HTML

Remove embedded HTML tags:

</p> pega </p> → pega

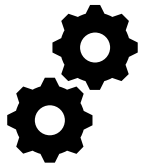
White Space

Remove all new line characters, tabs, etc.

Numbers

Generally remove all numbers

PRE-PROCESSING: TEXT



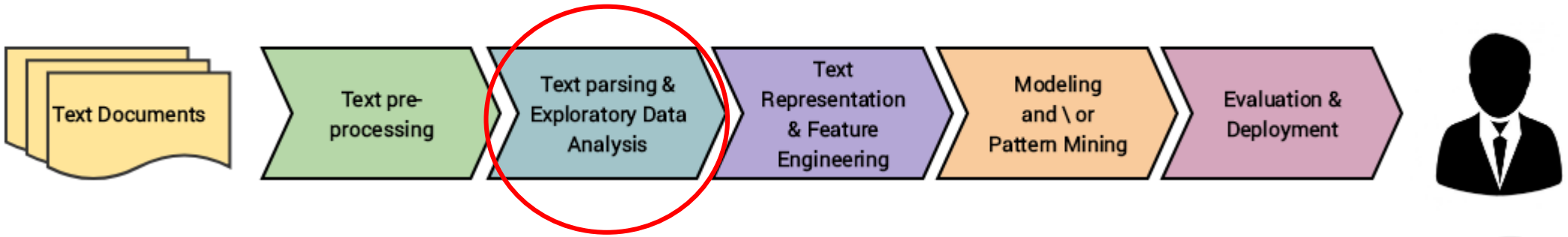
“SLA is not getting populated correctly – default SLA is 1 day”

preprocess_training_text()

“sla get populate correctly default sla day”

We created a standard python library of preprocessing functions to meet most NLP needs out-of-the-box

THE NLP METHOD



EXPORATORY DATA ANALYSIS

The Labels

The label set are the backlog ids. Each label in the set correspond to a group of samples. There are 158 unique backlog ids.

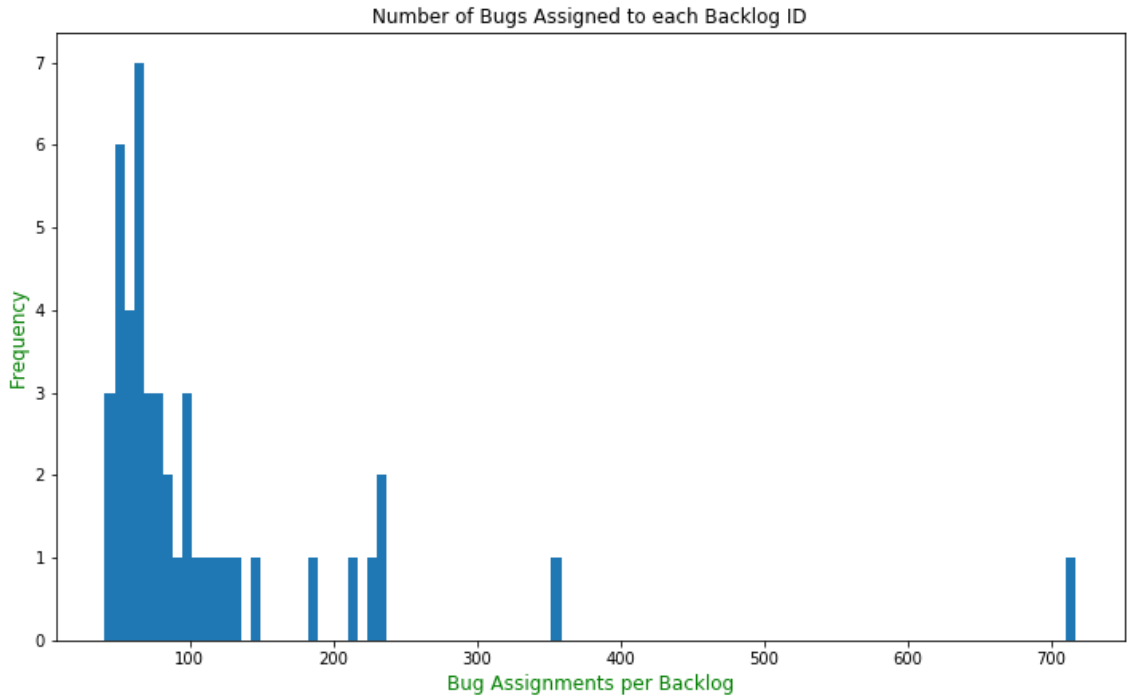
The Samples

The sample set is the bug description field concatenated with the subject field. Each sample in the set corresponds to a backlog ID. There are 6199 samples (unique description/subject fields).

The Training/Testing Split

60% of the data is used for training and 20% for validation, and 20% for testing

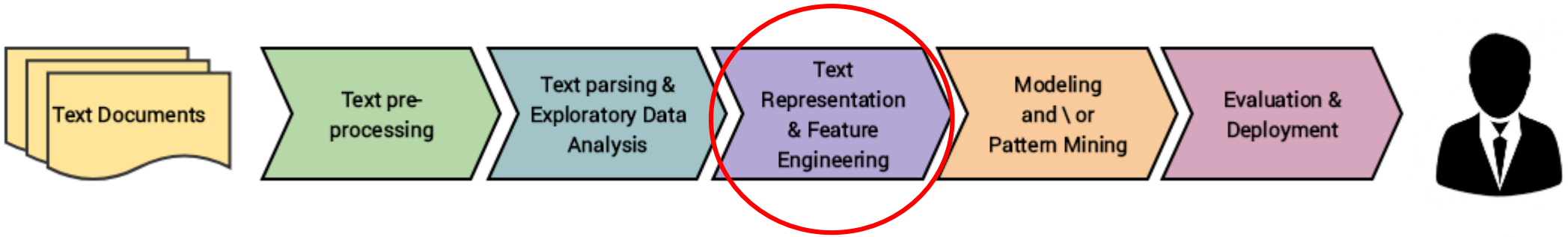
EXPORATORY DATA ANALYSIS



Eliminating Backlogs with less than 40 entries

Backlog ids with fewer than 40 bug entries had little predictive value, and models struggled to classify to them. 45 backlog IDs remain representing 85% of the sample set.

THE NLP METHOD



TEXT REPRESENTATION

Problem: Input of a machine learning model must be numerical. One cannot just feed words, documents, or text into a model

Solution: Text must be converted into some numerical representation that the machine learning model can understand

TEXT REPRESENTATION

Company Name	Price
VW	20000
Acura	10500
Honda	50000
Honda	10000

one_hot_encoding()

VW	Acura	Honda	Price
1	0	0	20000
0	1	0	10500
0	0	1	50000
0	0	1	10000

The Labels: One-hot encoding

The label set are the backlog ids. The backlog ids are encoded as a vector of size N, where N is the number of unique backlog ids. A dimension has a value 1 if it is the corresponding backlog id, else 0.

FEATURE ENGINEERING

Features: Information extracted from input data

Engineered Features: Information extracted from the input data to simplify the learning pattern between the input and output data.

Example:

$For\ input\ data\ X\ and\ output\ data\ Y\ and\ feature\ extraction\ transform\ T, T(X) = X' = xTx'\ ; learning\ function\colon\ xRy\ \rightarrow x'Sy$

Detailed explanation: The two mappings, $(X, Y)\ and\ (X', Y)$ represent the same pattern. In other words, a pattern can be represented equivalently using different mappings so we should chose a mapping for a given input that minimizes pattern recognition difficulty for a given model.

FEATURE ENGINEERING

Feature engineering approaches attempted to map the sample set:

1. Term Frequency – Inverse Document Frequency (TF/IDF)
2. Embedding
3. TF/IDF & Embedding
4. Hashing Algorithm

FEATURE ENGINEERING: TF/IDF

TF/IDF (Term Frequency-Inverse Document Frequency) : A measure of how important a word is to a document in a collection or corpus. Importance increases proportionally to the # of times a word appears in a document, but is offset by the frequency of the word in a corpus

How to compute:

$$\frac{Term\ Frequency}{Inverse\ Document\ Frequency} = \frac{\frac{\#\ times\ a\ word\ in\ a\ doc}{total\ \#\ words\ in\ doc}}{ln\left(\frac{\#\ docs\ in\ corpus}{\#\ docs\ word\ appears}\right)}$$

Example: In a document of 200 words, *pega* appears 10 times. We have 500 documents and *pega* appears in 250 of these.

$$TF = \frac{10}{200} = 0.05$$

$$IDF = ln\left(\frac{500}{250}\right) = 0.69$$

$$TF/IDF = \frac{0.5}{0.69} = 0.72$$

FEATURE ENGINEERING: TF-IDF

The TF-IDF Feature Set

TF/IDF was calculated for every word of the sample set i.e. all of the words for each label/description sample. This was used as the feature set

FEATURE ENGINEERING: EMBEDDING

Embedding: Text embedding is a measure of semantic similarity – it transforms every word in a corpus into a fixed-length vector representation where similar words are close to each other in vector space

Example: *"anchor"* and *"boat"* have similar embeddings but *"anchor"* and *"koala"* do not

EMBEDDING: CORPUS DEFINITION

Embedding Corpus Approaches: An embedding layer defines semantic similarity by modelling a finite corpus. There are typically two methods to define an embedding corpus.

1. Prebuilt embedding corpus: some form of pretrained embedding weights or collection of words ready out-of-the-box
2. Custom embedding corpus: a developer defined corpus used to train embedding weights. Often useful for unusual word combinations unique to the given corpus

The embedding corpus is used to train an embedding layer within the body of a machine learning model

EMBEDDING: CORPUS DEFINITION

Embedding Corpus Approaches:

Pre-built:

1. Google Hub Model
2. Google word2vec model

Custom:

1. Custom Pega Corpus

Combination:

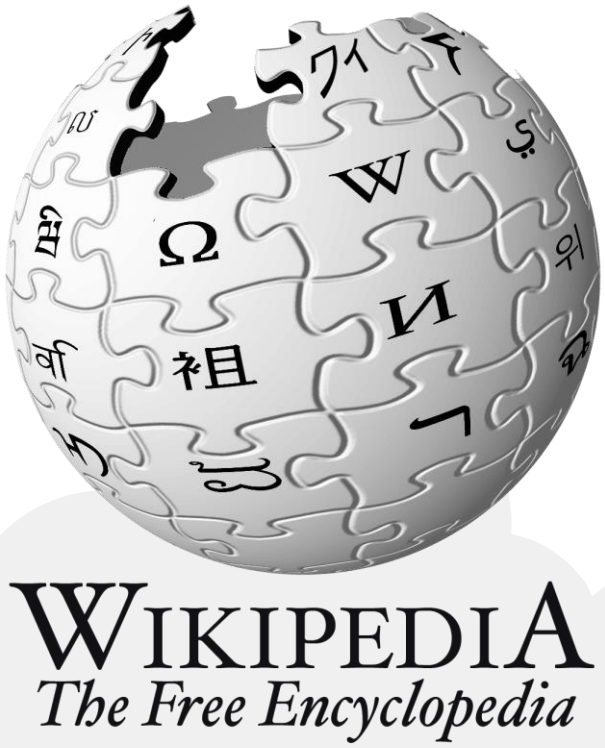
1. Pega Corpus + Brown Corpus

EMBEDDING CORPUS DEFINITION: TENSORFLOW HUB MODEL

TensorFlow Hub

Tensorflow Hub Model: Google offers pre-built models via Tensorflow Hub, a forum for the tensorflow AI/ML development platform. The hub model for embedding is trained on a corpus of 200B words from Google news, and can be imported as a layer that is added to a model. It takes an input of word tokens and weights each word appropriately using a doc2vec like technique explained later

EMBEDDING CORPUS DEFINITION: TENSORFLOW WORD2VEC CORPUS



Tensorflow word2vec corpus: Google offers a downloadable pretrained embedding matrix trained on millions of Wikipedia articles. This matrix is added to an embedding layer, and the developer can choose any of the embedding techniques described later in the presentation

EMBEDDING CORPUS DEFINITION: PEGA CRAWLER

Observation: Google's prebuilt corpuses were trained on news articles or Wikipedia articles and likely have limited exposure to Pega specific terms and associations.

Problem: Pega words like "infinity" and "fabric" infrequently have semantic similarity outside of Pega discussions but are often combined in Pega documentation and bug reports

EMBEDDING CORPUS DEFINITION: PEGA CRAWLER

```
result = word_vectors.similar_by_word("pega")
print('Most similar words:', result[:4])

Most similar words: [('crm', 0.7137225866317749), ('matrix',
0.6909430623054504), ('past', 0.6866768598556519), ('win',
0.6835222244262695)]
```

```
result = word_vectors.similar_by_word("trefler")
print('Most similar words:', result[:4])

Most similar words: [('ceo', 0.9964618682861328), ('founder',
0.9913451075553894), ('magazine', 0.9718469381332397), ('inc',
0.9717679023742676)]
```

Pega Crawler: Built a custom crawler that starts at the pega glossary and crawls through 281 links (single layer recursion) and pulls all relevant text (roughly 10500 sentences). These sentences are preprocessed and used to train a custom embedding layer. This embedding approach establishes seemingly accurate Pega associations.

EMBEDDING CORPUS DEFINITION: PEGA CRAWLER + AGILE STUDIO EPICS

```
result = word_vectors.similar_by_word("dennis")
print('Most similar words:', result[:4])

Most similar words: [('grady', 0.9726849794387817), ('q1',
0.9642074108123779), ('kielce', 0.9638020396232605), ('summer',
0.962353527545929)]

result = word_vectors.similar_by_word("pega")
print('Most similar words:', result[:4])

Most similar words: [('pega7', 0.6667320132255554), ('premise',
0.5724167823791504), ('crm', 0.530005931854248), ('unified',
0.5297926664352417)]
```

Pega Crawler + Agile Studio Epics:

Combined the corpus built using the Pega crawler with scraped sentences from Agile Studio. This significantly increased the size of the corpus (47068 sentences)

EMBEDDING CORPUS DEFINITION: PEGA CRAWLER + AGILE STUDIO EPICS + BROWN CORPUS

```
result = word_vectors.similar_by_word("cloud")
print('Most similar words:', result[:4])

Most similar words: [('environment', 0.9660835266113281),
('deploy', 0.958573579788208), ('provision',
0.9243481159210205), ('infinity', 0.9242792129516602)]


result = word_vectors.similar_by_word("app")
print('Most similar words:', result[:4])


Most similar words: [('customize', 0.9222714900970459), ('sfdc',
0.9199884533882141), ('device', 0.9197338223457336),
('interface', 0.9181100130081177)]
```

Pega Crawler + Agile Studio Epics + Brown Corpus:

Combined the corpus built using the Pega crawler and scraped sentences from Agile Studio with the Brown Corpus. The Brown Corpus is from 500 samples of English-language text, totaling 57340 sentences. This intended to generalize the Pega corpuses.

DOCUMENT EMBEDDING: APPROACHS

Document Embedding Approaches:

We took several approaches described in NLP literature to convert our emails (documents) to numerical data while retaining meaningful information.

1. Standard Embedding
2. Greater-than-word-length embedding
3. Recurrent Neural Networks

EMBEDDING: DOCUMENT EMBEDDING

Document Embedding

In all cases, embedding layers are used to convert text into a *numerical* feature-set. By representing similar words as similar vectors, we can retain the informational value in the input text.

It is straightforward to convert a single word into a numerical vector, however, retaining the informational value of an entire *document* can be challenging.

Understanding a document requires one to retain the **meaning of each of the words in the document**, and the **relationships between those words**.

EMBEDDING: DOCUMENT EMBEDDING

The Problem: Capturing the multitude of words in a document and the relationships between them in a numeric format is challenging:

Grammar structure of a sentence contained in our bug dataset:

DOCUMENT EMBEDDING APPROACHES: STANDARD EMBEDDING

Standard Embedding, in essence, simply represents a document as a list of words. It uses the embedding layer to generate the numerical representations of each of the words in a given document.

Example:

Input: "pega app studio shuts down randomly"

Output: [Vector 11, Vector 76, Vector 264, Vector 2956, Vector 207, Vector 11915]

In the above case, vector 11 represents the word "pega," vector 76 represents the word "app," and so on. We then pass all of these vectors to our ML model to learn from:

Vector 11: [1.89106214e+00, -1.20326054e+00, 2.14849383e-01, 7.65558720e-01, 1.14027750e+00, -4.96039182e-01],

Vector 76: [-7.19669437e+00, -1.05293906e+00, 3.96388936e+00, -4.03765011e+00, -7.04570234e-01, -2.72737533e-01],

Vector 264: [1.41667485e+00, -4.29529756e-01, 3.52789164e-01,-3.41378927e+00, -2.51580167e+00, 5.22404814e+00],

Vector 2956: [-1.14397550e+00, 1.44761801e-01, -7.16003552e-02,-1.24786353e+00, 1.25313675e+00, -5.06561160e-01],

Vector 11915: [6.89014018e-01, -1.24445379e+00, -8.23560774e-01, 6.23422682e-01, 6.76878154e-01, 2.23589405e-01,]]

Vector 207: [-1.55395925e-01, -3.61439824e-01, -2.47564569e-01, -3.94252747e-01, 4.99546260e-01, 7.47200549e-02],

DOCUMENT EMBEDDING APPROACHES: STANDARD EMBEDDING

Advantages:

- **Easy** to implement
- Does not throw away information about individual words in the document (**no compression**)

Disadvantages:

- **Lacks information about the relationships between words** in a document and **the grammatical structure** of the document
  - Example: A “technical bug” is quite different from a “June bug”, but standard embedding represents “bug” in the same way for each of these examples.
- **Results in an extremely large number of features**. Largest email has 295 words * 300 embedding vector dimensions = **88.5K features**

DOCUMENT EMBEDDING APPROACHES: GREATER-THAN-WORD-LENGTH

Greater-Than-Word-Length Embeddings combine the embedding vectors of each of the words in the document. It outputs a *single* vector that represents the document itself.

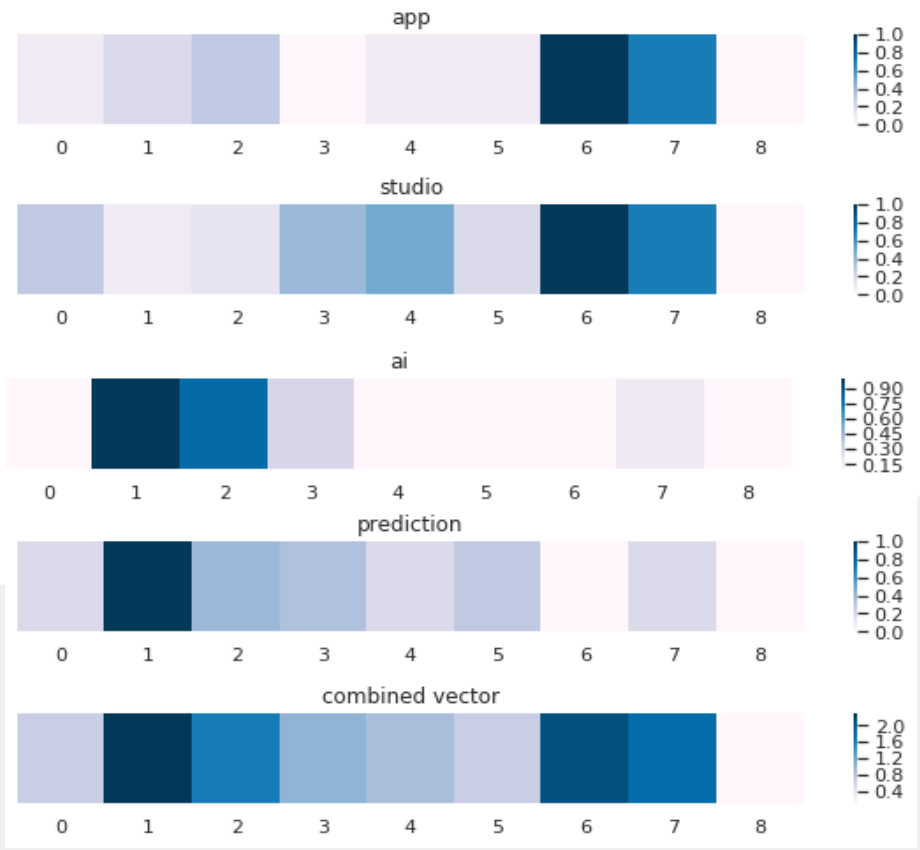
Example:

Input: "pega app studio shuts down randomly"

- **Add** Vector 11, 76, 264, 2956, 207, 11915 all together
- **Normalize** the resultant vector

Output the combined vector : [-0.3247, -0.07167, 0.1781 , -0.3731 , -0.7791]

DOCUMENT EMBEDDING APPROACHES: GREATER-THAN-WORD-LENGTH



In the example at left, multiple word vectors are summed together and converted into a single "combined" document vector. You can see that the words "app" and "studio" are represented quite similarly, and that "prediction" and "ai," likewise, are represented quite similarly.

If a model took in the words "app" or "studio" independently, it might predict the bug belongs to App Studio. Likewise, if a model took in the words "ai" or "prediction" independently, a model might predict the bug belongs to customer decision hub. **The combined vector, however, shows us that the document in question has several words related to both "app" and "AI."** Using this document vector, a machine learning model might correctly predict that a bug belongs to **prediction studio.**

DOCUMENT EMBEDDING APPROACHES: GREATER-THAN-WORD-LENGTH

Advantages:

- **More explicitly encapsulates information about the whole document** than individual word vectors do.
- **Draws inferences based on word groupings** (app-related words + AI-related words = Prediction Studio bug)
- **Few features** = less computationally intensive and easier to draw insights

Disadvantages:

- **Does not retain grammatical structure** or sequences
- Compressing word vectors into a single document vector **results in data loss**

DOCUMENT EMBEDDING APPROACHES: RECURENT NEURAL NET (RNN)

RNN Embedding: Recurrent neural networks form directed cycles from internal states – meaning they can establish relationships among sequences. LSTMs (long short-term memory) recurrent networks can therefore discover patterns that normal document based embeddings cannot.

Example: ***"At the factory, the supply chain manager app is malfunctioning"*** and ***"The app factory is malfunctioning"*** are bugs that should likely be classified to different backlogs. Other document embedding approaches likely see words app and factory and assign similar vectors in both cases. **RNN embedding recognizes the difference in word sequence – establishing distinct vectors that more accurately represent the respective semantic intentions**

DOCUMENT EMBEDDING APPROACHES: RECURENT NEURAL NET (RNN)

Advantages:

- **Draws inferences from sequences –** Able to recognize more meaningful patterns and therefore increase accuracy in certain use cases

Disadvantages:

- **Hard to implement**
- **Requires that given sentences follow standard structure and sequence** - this is not always the case with email subjects and bodies

TF/IDF & Embedding

TF/IDF & Embedding: The embedding vector outputted by the embedding layer for the different approaches was concatenated with the TF/IDF vector, making the feature space both the TF/IDF vector and the embedding vector. In theory, the two together may extract additional insights that were not readily apparent individually.

HASHING ALGORITHM

Hashing vectorizer: The hashing vectorizer applies a hashing function to the term frequency counts in each document, efficiently mapping words to a vector of specified size.

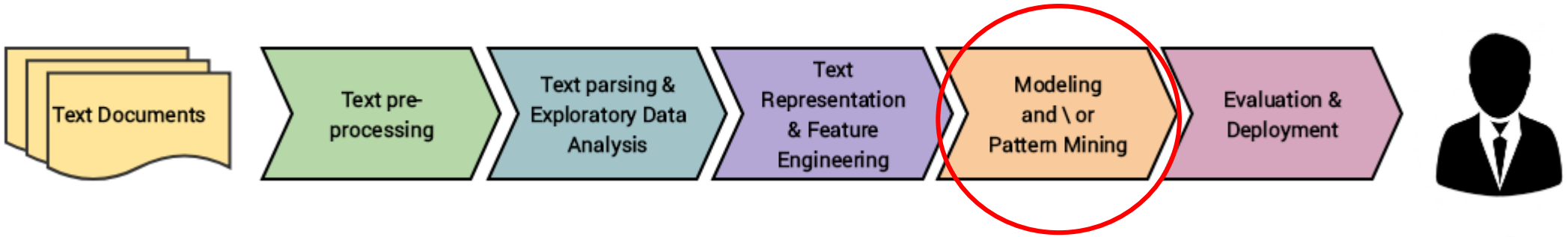
Advantages:

- Low memory and scalable to large datasets
- Can be used in parallel pipeline as there is no state computed
- Easily implementable

Disadvantages:

- Hashing collisions can map the different samples to the same feature (this can be a disadvantage or advantage depending on the characteristics of the dataset)
- There is no IDF weighting (making the vectorizer stateless) which can negatively impact performance

THE NLP METHOD



THE MODEL: DEEP LEARNING

Deep Learning: Deep learning extracts meaningful patterns from the feature set, making connections between the input and the output. Using multiple layers, deep learning progressively extracts higher level features inferred from the input, making better connections to the output and becoming more capable of making accurate inferences.

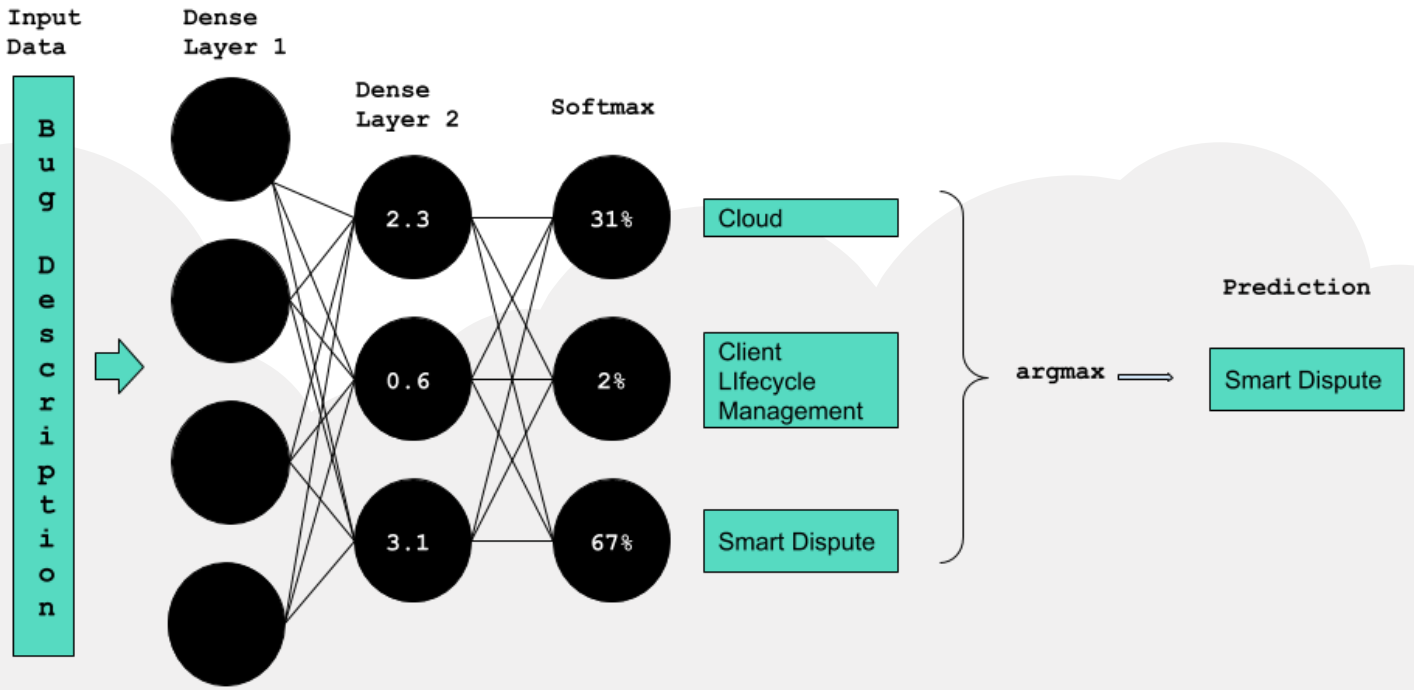
Why:

"for most flavors of the old generations of learning algorithms (artificial neural nets) … performance will plateau … deep learning … is the first class of algorithms … that is scalable … performance just keeps getting better as you feed them more data"

- Andrew Ng, Founder of Google Brain

PREDICTIONS AND MODEL CONFIDENCE

For multi-class classification problems (i.e. assigning emails to one of many possible backlogs), DNNs use a SoftMax output. A SoftMax output is a probability distribution that shows the likelihood that a given example belongs to each given class:



EVALUATING MODEL CONFIDENCE

Deep Learning Models are able to perform powerful pattern recognition tasks, allowing for automation of rote tasks and powerful prediction capabilities beyond the scope of human capability. **Model predictions, however, are often taken blindly and assumed to be accurate.**

Developing model confidence estimates is crucial to preventing catastrophic errors caused by deep learning. **But teaching a model to "know" its limitations**, as it turns out, **is easier said than done**.

EVALUATING MODEL CONFIDENCE

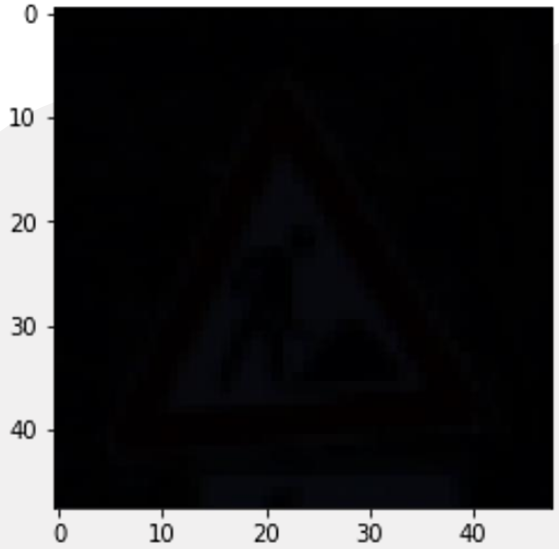
The Key Issue is this: models are trained to make predictions. Under normal circumstances, models do not have the option to place a piece of data into an "unknown" category. As such, DL models will attempt to make a prediction, even when the input data is poor or unusual.

Predicted Label: Road Work

99% Confident

Predicted Label: Roadwork

92% Confident



In the example at left, we have a DL model that is trained to look at road signage and classify the type of sign on the road.

When we feed it the left sample, it correctly predicts the sign as "Road Work."

When feeding it the dark image at right, however, it forces a prediction even though the input clearly is not a sign at all, and incorrectly estimates a high confidence value.

EVALUATING MODEL CONFIDENCE

As we could see in the former example, SoftMax probabilities do not reflect model confidence.

We can, however, determine model confidence using a simple rule:
`Confidence is Consistency`

- In the testing phase: **evaluate the model for accuracy**
- In production: **Evaluate the model for precision**

EVALUATING MODEL CONFIDENCE: TESTING FOR CONSISTANCY

- A neural net is just a complicated algorithm. As such, **under normal circumstances, the same input always produces the same output** (2 + 2 = 4, no matter what)
- To test precision, however, we want to see **what happens if we tweak the input data, just a little bit, or tweak the way that our machine learning algorithm works**.
- After each tweak, we check whether or not the model makes the same prediction, or a different one. **If we make 10 tweaks and our model makes the same prediction 10 times, our model is confident. If we make 10 tweaks and our model produces different predictions after each tweak, our model is unsure.**

EVALUATING MODEL CONFIDENCE

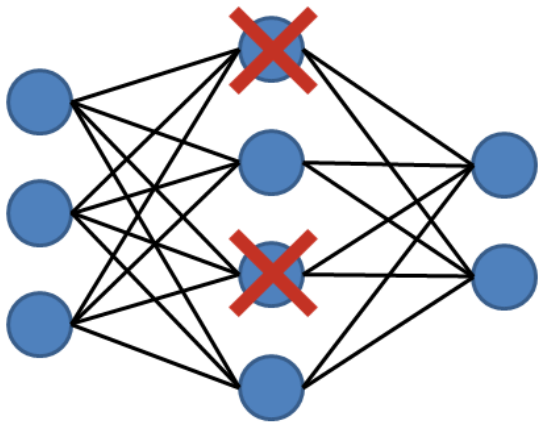
The Paper, "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning" (2015) describes a clear way to make these types of "tweaks."

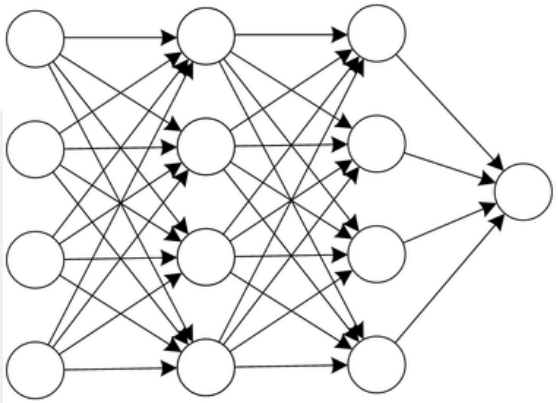
The approach is termed "Monte Carlo Dropout"

EVALUATING MODEL CONFIDENCE

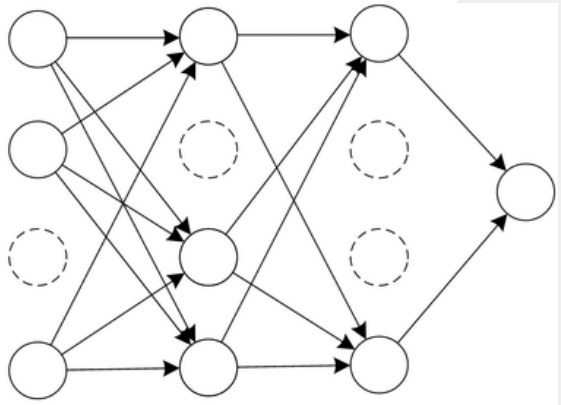
Dropout:

- Dropout is an approach developed by Google (2014) to **prevent overfitting of training data**.
- Dropout causes a certain proportion (say 30%) of the nodes in a deep net to go “silent.”
- **By depriving the model of information, we ensure it does not rely too heavily on any one node in the network or any one piece of the input data.**





(a) Standard Neural Network



(b) Network after Dropout

EVALUATING MODEL CONFIDENCE

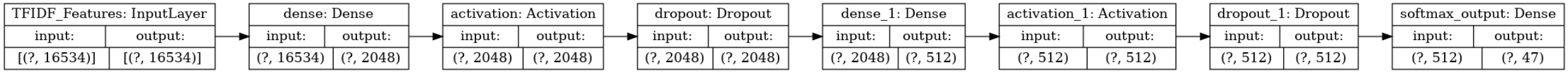
Monte-Carlo Dropout:

- Monte-Carlo dropout selects nodes and features of the input data **at random**, and **drops out different features each time it is applied**.
- Typically, Dropout is only applied during **training** to prevent models from overfitting training data. But **Monte-Carlo Dropout is applied during both training and testing.**
- To test the "consistency" or "precision" of the algorithm, we pass in the same sample multiple times, applying different dropout schemas each time. **If the model makes the same prediction no matter which dropout schema is applied, the prediction is robust and confident.**

Confident?	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Yes	Backlog 1	Backlog 1	Backlog 1	Backlog 1	Backlog 1
No	Backlog 1	Backlog 5	Backlog 1	Backlog 4	Backlog 4

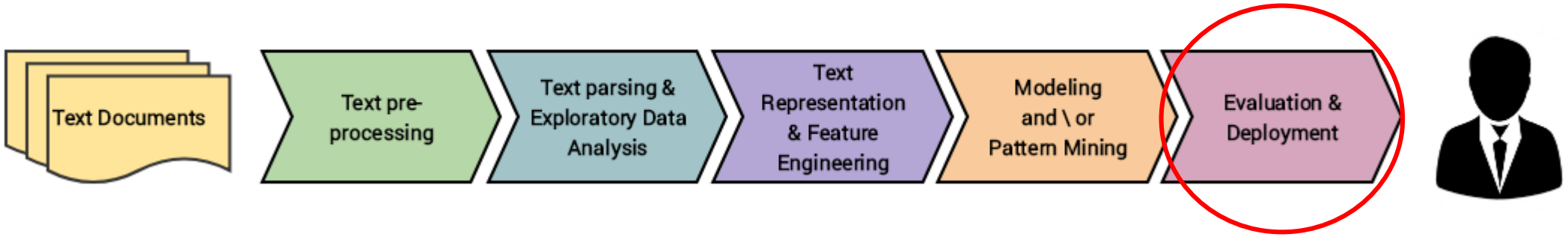
EVALUATING MODEL: SUMMARY

The following is a diagram of the overall model and a chart of the basic characteristics:

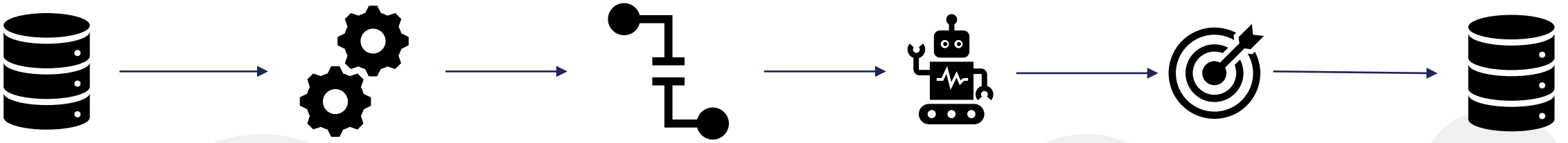


Activation	Optimizer	Loss Function	Output	Input
Tanh	Adam	Kullback Leibler Divergence (outperformed Categorical Crossentropy)	SoftMax	TF/IDF Features

THE NLP METHOD



EVALUATION: OVERALL ARCHITECTURE



All Pega bug submissions with description and label

Text preprocessing

Feature Engineering

Model

Confidence Bounding

Classifications

EVALUATION: PERFORMANCE COMPARISON

Key:

- **Pega Corpus** = Pega Website + Agile Studio Epics
- **Combined Corpus** = Pega Website + Agile Studio Epics + Brown Corpus
- **Zero Rule Algorithm** – always predicts the class value that is most common in the dataset. i.e. if BL55 is the backlog with the most bugs, the zero rule algo. classifies all emails to BL55. Zero Rule is a stronger assessment of baseline accuracy than a random selection algorithm.

Approach	Validation Accuracy
TF/IDF + Dense Net	80%
Hashing (TF only) + Dense Net	72%
Pega Corpus Embedding w/ **LSTM**	67%
Combined Corpus Embedding w/ **LSTM**	66%
Google Hub Model (Universal Sentence Encoder – advanced doc embedding)	60%
Combined Document Embedding	40%
Pega Corpus Document Embedding	41%
Zero Rule Algorithm (Baseline)	18%
Random Selection (Baseline)	3%

EVALUATION: PERFORMANCE WITH CONFIDNECE

Capable of predicting 100% of all bugs with 80% accuracy using TF/IDF approach

Capable of predicting 70% of all bugs with 90% accuracy using TF/IDF approach

Capable of predicting 55% of all bugs with 94% accuracy using TF/IDF approach

Capable of predicting 30% of all bugs with 99.1% accuracy using TF/IDF approach

EVALUATION: CONFIDENCE BOUNDING