# Clustering Enhanced Merged-Average Classifier via Hashing

*Shubham*

Master of Science
School of Informatics
University of Edinburgh
2020

# Abstract

This projects aims at improving an existing technique for extreme classification called Merged-Average Classifier via Hashing or MACH. MACH was able to successfully reduce the linear space complexity (in number of classes) involved in extreme classification to sub-linear levels by splitting the classification problem into smaller sub-problems that were easily parallelizable. At prediction time, MACH has to perform decoding which comprises of two tasks: computation of score for all the classes and finding top-k classes with the best scores. Our aim is to reduce the time required for decoding by pre-filtering the top classes before computing the final class scores. We propose two techniques, N-MACH and C-MACH, to meet the goal of this project.

## Acknowledgements

I express my heartfelt gratitude to my supervisors Prof. Iain Murray (School of Informatics, The University of Edinburgh), Dr. Sean Murphy (Amazon, Edinburgh) and Dr. Gianluca Corrado (Ex-Amazon, Edinburgh) for their constant guidance, insightful discussions and positive criticism which played an important role in completion of this project. Without the encouragement from my supervisors, this dissertation wouldn't have been possible.

I fall short of words to express how obliged I am to my parents Dr. Satya Shila Singh and Dr. Arun Kumar Mishra who have always supported me since my childhood in all my endeavours and without whom, I would have never reached this milestone in my life. My parents have been a constant source of motivation for me.

I also extend my gratitude to my friends Mr. Indraneel Brahma, Mr. Kurtulus Bulus, Mr. Sudipta Saha, Mr. Vedant Agrawal and Mr. Zhongyu Lang who have always been there for me.

I extend my special gratitude to Ms. Sakshi Tayal who has always supported me irrespective of any situation. She has always supported me in following my aspirations. Without her, it wouldn't have been possible for me to finish this dissertation.

Lastly, I would like to thank The University of Edinburgh, the staff of the Informatics Teaching Organisation and the staff of the Student Support Services for providing me and my peers with such a great infrastructure and positive environment .

# Table of Contents

# Chapter 1

# Introduction

Over the years, the user activity on internet has grown a lot leading to a rapid growth of websites like Amazon [1], Wikipedia[2], Reddit[3], Netflix[4] and many more. This has turned websites like these into great resources for data. The data stored, generated or collected by any website only tends to become more and more complex with the passage of time. Here, when we say that data is becoming complex, this is with respect to different aspects of data which make it hard for a user to handle the data or use the data in some process/task, e.g. the sheer size of data can make storage a concern when the resources are limited, noise associated with data requires a user to perform extra steps for data cleaning before it can be used in any process, etc . Such complexities in data are a major challenge for researchers who use this data in machine learning research. A really great outcome of such challenges faced by researchers is that a lot of progress has been made in different areas of machine learning. One such area of machine learning is the one that deals with Multiclassification problems.

The goal of multiclassification is to take an input and predict the corresponding output label(s). Multiclassification problems can take several forms such as character-recognition [31], parts-of-speech tagging [12], text-categorization [29], etc. Multiclassification problems can be of two types: Multi-Class Classification [2] and Multi-Label Classification [34]. In Multi-Label Classification, each data point can be associated with one or more than one labels at the same time where as, for Multi Class Classification, each data point can only be associated with one label.

As the data became more complex, the simple task of Multiclassification also

[1]`https://www.amazon.com/`

[2]`https://www.wikipedia.org/`

[3]`https://www.reddit.com/`

[4]`https://www.netflix.com/`

started to become more difficult. *Extreme Classification* is a great example for that. Extreme classification is a multiclassification problem in which the dimensionality of the label space is really huge, usually in millions. Such high dimensionality of label space tends to create a huge memory requirement by the last layer (which is usually a softmax layer) of the classifier.

Recently, research community has been showing a great interest in extreme classification due to its potential to be used in the industry for tasks like information retrieval. As a result of this, a lot of research has been carried out to deal with the statistical and computational challenges posed by extreme classification problems. Most of these works tried to propose different ways to reduce the linear time and memory complexity in the number of classes to sub-linear levels. One such technique is Merged-Average Classifiers via Hashing or MACH [23]. MACH achieves this by breaking down the classification problem into several parallelizable sub-problems. Even though MACH has demonstrated great performance with respect to memory requirements and training time compared to other extreme classification methods, MACH still takes time linear in number of classes to predict the top labels at the test time.

In our research, we propose a versions of MACH called Clustering Enhanced Merged-Average Classifier via Hashing (C-MACH) which uses label [5] clustering as a solution to overcome MACH's linear time complexity at decoding time (decoding refers to task of calculating score for every label followed by finding the top-k labels). In addition to this, we compare C-MACH with a naive approach called N-MACH or Naive-Merged-Average Classifier via Hashing as it naively utilises the existing structure of MACH for reducing the decoding time. The motivation behind this research is that in today's world when machines are operating at latencies in microseconds while dealing with thousands of requests in a second, even a small amount of time saved makes a lot of difference. Consider the example of Amazon Product Search, if Amazon Product Search is able to save some time on a user query, it means that it can serve more users in the original amount of time.

The rest of the research is organised into the following chapters:

- **Background**: This chapter builds up the context of our research and throws light on the progress made towards extreme classification so far by the research community.
- **Methodology**: This chapter gives details about MACH and our proposed versions of MACH: C-MACH and N-MACH.

[5]In this document words *label* and *class* are used interchangeably

- **Experiments and Evaluation**: This chapter gives specific details about how the experiments were organised. This chapter also gives an insight into the observations made from the experiments that were conducted by us.

- **Discussion**: This chapter is associated with the interpretation of the results obtained from our experiments.

- **Conclusion and Future Work**: This chapter summarises our project and lays some possible areas in which our work might be extended in future.

# Chapter 2

# Background

In recent times, a lot of work has been done in the area of extreme classification. Most of the work that has been done towards developing new extreme classification techniques has been mainly focused towards four different approaches, namely: 1-vs-all, label embedding, tree and deep learning based approaches.

1-vs-all approaches, as the name suggests, make use of one classifier per label. In other words, if there are $K$ classes then there will be $K$ classifiers where $k^{th}$ classifier's output represents how likely a data point belongs to the $k^{th}$ class. Scores from all the classifiers are compared to find the most likely classes. Generally speaking, most of the methods that belong to this category have been seen to perform better than their counter parts in other three approaches. Even though methods under this category perform really well, requirement of specialised hardware for distributed training of classifiers is a downside. Slice [16], Parabel [27] and PPDSparse [41] (which is an improved and parallelized version of PDSparse [42]) are some of the most successful one-vs-all approaches. All three of these 1-vs-all approaches utilise a negative sampling method so that the training process doesn't require all the training points but only a portion of it. The negative sampling approach reduces the training time significantly. Out of the three approaches, Slice has the upper hand because it is the only one out of three that can be trained on low-dimensional dense features. These low-dimensional dense features can easily be obtained from weights of hidden layers of pretrained deep networks . Some of the other approaches that use the 1-vs-all method are [4], [5] and [19].

Under label embedding based approaches, the entire label space is projected to a lower dimension latent space. Some of the ways in which this is achieved is by using compressed sensing [15], binary relevance [43], bloom filters [11], etc. Most of these approaches tend to work fine when the label dimentionality is close to magnitudes of hundreds but when the label dimentionality exceeds that threshold, the performance starts to degrade, it's because of the low rank assumption that forms the basis of such approaches[39]. This low rank assumption doesn't work with datasets that have a label distribution based on power-law, i.e. existence of tail labels in the label distribution, which is usually the case with huge datasets [19]. SLEEC [6] is an embedding based approach to extreme classification which tries to mitigate this issue by not relying on the low-rank assumption, it achieves this by learning a label space embedding which preserves the correlations between labels. This approach was successful because instead of using linear compression, embeddings use non-linearity to compress the label space by virtue of which embeddings are able to maximize the information retained. W-LTLS [14], another embedding based approach, tries to learn random binary label embeddings and at prediction time, uses loss based decoding to find the most probable label for multi-class datasets in sub-linear time complexity in label dimentionality. A great advantage of W-LTLS is that it provides an ability to the user to do a trade-off between training/prediction time and the accuracy. [32], [35], [37] are some of the other extreme classification approaches that are based around label embeddings.

Under tree based extreme classification approaches, the label space is recursively partitioned . The partitioned label space makes it possible to carry out training and prediction in sub-linear time complexity with respect to the number of classes. Tree based approaches usually scale really well with huge label space. Despite its advantages, tree based approach approach suffer with a big set back which is it's poor performance on tail labels due to error propagation [17]. Similar problem occurs with classification schemes that utilise a hierarchical softmax [3]. LOMTree [10] is one such approach, it converts a label space into binary problems which is then used to convert the problem into a tree hierarchy. The tree produced through binary splitting is balanced. The splitting process in LOMtree is such that it ensures splits are balanced and pure. FastXML [28] takes a slightly different approach, instead of splitting the label space into a tree hierarchy, it splits the feature space at each internal node into two. At each internal node of the constructed feature space tree, a linear classifier learns this split. PfastreXML [17] is an improved version of FastXML. PfastreXML is able to improve on tail label prediction by using a set of specific classifiers that rank the tail-labels and by replacing the loss function of FastXML with one which is based on propensity.

Approaches based on deep learning rely heavily on word-embeddings to transform the input feature space of the data on which they are trained. Deep learning based approaches usually perform worse compared to the other three approaches as there aren't enough instances that represent tail labels. One of the most cited researches in this domain is FastText [18]. FastText was designed for document-classification task with a huge label space. The idea was to compute the word-embedding for every word that lies in a document and then calculate their average. This averaged embedding represents a document. This transformed feature space for each document is then used as input to a linear classifier with a soft-max layer which learns to classify the document. The drawback of this approach is that due to averaging of the embedding the information about the order of words in a document is lost. In order to incorporate word order, a technique called XML-CNN [22] was developed. XML-CNN uses a moving convolution window , just like an image classification CNN [20], over the word embeddings of a document followed by an implementation of a dynamic max-pooling layer [9].

In addition to above mentioned approaches to extreme classification, a very recent work by [23] tries to mitigate the problem of huge memory requirements in extreme classification caused by the outer most layer of a classifier. It was called MACH or *Merged-Average Classifier via Hashing*. MACH tries to split the classification problem into several smaller classification sub-problems. These sub-problems can easily be handled by training a different classifier for each sub-problem. A major advantage of this scheme is that all the classifiers can be trained in an embarrassingly parallel fashion. Once all the classifiers are trained, during prediction time, predictions from all the classifiers are combined to deduce the final result. MACH is able to deal with memory issue posed by extreme classification problem but it still leaves one issue unsolved: during the process of predicting the correct labels, prediction algorithm has to compute scores for every class/label and then find the most probable labels. This part of prediction stage will be referred as *Decoding* in our research. Therefore, MACH still takes time linear in number of labels ( or linear in label dimentionality) for the process of decoding. A more detailed explanation of MACH's working mechanism have been dealt in section 3.2. Our aim is to improve on this aspect of MACH, i.e. reduce the time required for decoding.

# Chapter 3

# Methodology

## 3.1 Preliminaries

Before we start off with the description of the work carried out, let's first understand the notion of different symbols used throughout this document. In this document, [] has been used to represent a range of integers, eg. [5] represents $1, 2, 3, 4, 5$. For representation of data, symbol $D$ has been used where $D = (x_i, y_i)_{i=1}^{N}$, $x_i \in \mathbb{R}^d$ and $y_i \in \{1, 2, 3..., K\}$. $d$ represents the dimensionality of the input points, i.e each data point or $x_i$ has $d$ features. $K$ is the number of classes in the data. $(x, y)$ will represent a generic datapoint.

## 3.2 Merged-Average Classifier via Hashing

As mentioned earlier, MACH solves the problem of huge memory requirement of extreme classification caused by the massive dimensions of the last layer of neural networks by splitting a big classification problem into multiple smaller classification subproblems. This results in a memory requirement of $O(logK)$ instead of $O(K)$. The pipeline for MACH, from training to prediction, involves following stages:

### 3.2.1 Partitioning

At this stage, the classification problem is split into several smaller classification problems. The idea is to make random groups of class labels and then repeat the process several times such that no two classes ever fall in the same groups for all the repetitions. This is achieved by means of *2-Universal Hashing Function*. For some function

$h : [f] \rightarrow [B]$ to be considered as a 2-universal hash function, it should have the property that every $i, j \in [f]$ where $i \neq j$ and for any $w_1$,$w_2 \in [m]$:

$$P(h(I) = w_1 \ \&\ h(j) = w_2) \ = \ \frac{1}{B^2} \tag{3.1}$$

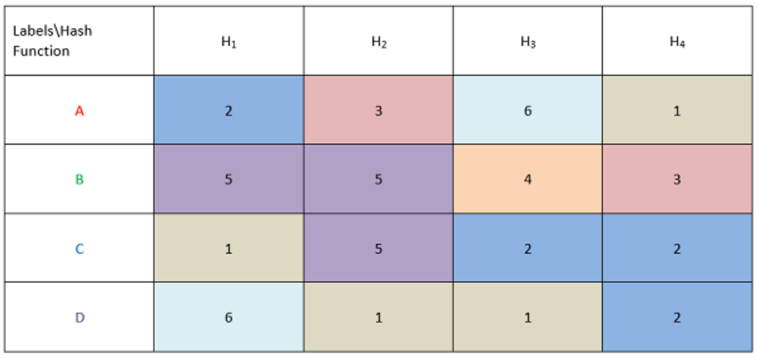
[7] showed that one of the easiest ways to create a 2-universal hash function is by taking a random prime number $p$ such that $p \geq B$. This is then followed by use of $[0, p]$ as the sample space to draw two random numbers $a$ and $b$ which are then used to compute hash values as follows:

$$h(x) = ((ax+b) \ mod\, p) \ mod\, B \tag{3.2}$$

Now, in our case, $f$ is replaced by $K$ and $B$ is a number much smaller than $K$, usually in some magnitude of $logK$. Hence, the 2-universal hashing function maps from $[K]$ to $[B]$, i.e $h : [K] \rightarrow [B]$. Several of these 2-universal hashing functions with different signatures are used to hash the labels in the data. So, essentially, now we end up with $h_i : [K] \rightarrow [B]$. $[B]$ represents the possible buckets or meta-classes to which a label can be hashed to. Buckets can also be referred to as meta-classes. Since, the signature of every 2-universal hashing function is different for the different values of $i$, this results into each label being hashed into a different bucket or meta-class for different 2-universal hashing functions. Figure 3.1 represents how four classes ($K = 4$) $A$, $B$, $C$ and $D$ are hashed into one of the possible six buckets ($B = 6$) by using four different 2-universal hashing functions. This results into $h_i : [4] \rightarrow [6]$ ,where $i \in [4]$.

### 3.2.2 Training

Let's say that we have $R$ *2-Universal Hash Functions* functions. This means that we end up with a total of $R$ different ways in which a class label can be hashed. Therefore, we train $R$ different models on the data. For simplicity, each model will be referred as the $r^{th}$ repetition where $r \in [R]$. The process of training involves use of neural networks which train on the data using back-propagation. Back-propagation tries to minimize the loss by adjusting the weights of a neural network. Loss is given by a loss function. Neural networks used in MACH try to minimize Binary Cross Entropy loss [34] when learning on a multi-label data and Categorical Cross Entropy loss [2] when learning on a multi-class data.



| Labels\Hash Function | H1 | H2 | H3 | H4 |
|---|---|---|---|---|
| A | 2 | 3 | 6 | 1 |
| B | 5 | 5 | 4 | 3 |
| C | 1 | 5 | 2 | 2 |
| D | 6 | 1 | 1 | 2 |

Figure 3.1: An example illustrating how classes hashed by using different hash functions. Even though this example has a $B > K$, it can easily be understood how with just $B = 2$ the four classes can easily be hashed. With $B = 2$, $2^4$ classes can be represented such that no two classes have the same signature. In fact, with $B = 2$ and with just 2 universal hash functions four classes can be given different signatures (because $2^2$), the only thing being that now there would be a higher probability of clashes between classes' hash signatures.

The architecture of neural network is kept constant for each of the $R$ repetitions. Also, the input of the every neural network is the same, the only thing different for each neural network is the $y$ value that is fed to it. The $y$ value for $r^{th}$ repetition is the hashed value of true label which is represented as $h_i(y)$. A key thing to note about the training process is that all the $R$ repetitions can be trained in embarrassingly parallel fashion which means that if the average training time for each repetition is $t$ then the total training time is not $Rt$ but rather just $t$. This type of no-communication parallelism is the key for MACH's really low training time.

### 3.2.3 Prediction

This stage is the one in which data (test) is fed to all the $R$ trained neural networks for forward propagation. The output of this stage can either be soft-max probabilities or raw logit values from $R$ trained model repetitions. In our experiments, raw logit values were used as it was reported in [23] that logit values were the one that performed better. For simplicity we will refer to these predictions as meta-class scores. If we assume that the number of samples or data points used in this phase were $n$ then, each repetition will produce an output (meta-class scores) of size $nB$. If $B << K$ then, memory requirement is $nRB$ which is less than $nK$. As an example, if $K = 30,000$,

$B = 16$ and $R = 16$, then $16(512)n = 8192n < 30000n$.

### 3.2.4 Score Estimation

The final estimate of the scores represents which labels are more probable for a particular data input. This means that a label which has higher score is more likely than a label with lower score. In order to compute final score of a class $l$ for a particular data input, the individual scores of each meta-class (into which the class $l$ was hashed by using $R$ 2-universal hash functions) are combined. In the original MACH research [23], average, min and max estimators were used for this purpose. An example for average estimator can be explained by means of figure 3.1, final score of class $A$ for a generic data point $x$ is given by: Final Score for class $A = \frac{\sum_{r=1}^{4} Score(H_r(A))}{R}$, where $R$ is 4. In the given equation, Score represents the individual score from each individual repetition. For min and max, the lowest and the highest individual scores are used as final score, respectively. Though, in our research, we have used average or the unbiased estimator which has been reported to have the best overall performance.

### 3.2.5 Finding the top-K Labels

This is the final stage of the MACH. For each data point, argmax has to be performed on the computed scores for all the labels when the data is multi-class. In case of multi-label data, argsort has to be performed in order to find the top-k label.

## 3.3 Effectiveness of MACH

At the score estimation stage of MACH (section 3.2.4), final scores for $n$ data points are computed. The process of computing the final scores involves computing scores for all the $K$ labels by combining their meta-scores using one of the estimation schemes. This results into a time complexity which is linear in $K$ for this stage. Since the dimensionality of final score matrix is in $K$ for each instance, the process of finding the top-$k$ labels is also linear in $K$ with respect to the time complexity. Therefore, MACH successfully manages to reduce the memory complexity created by extreme number of labels but it isn't able to handle the linear time complexity induced by the aggregation of sub-problems at the prediction-time. This is the aspect which we intend to improve on in this project.

### 3.3.1 Improving on MACH

A Naive approach to improve MACH's linear time complexity involved in computing the class scores for $K$ classes would be to take meta-class scores from one of the already trained $R$ repetitions of MACH and then consider only those classes $k$ that fall in top-$b$ buckets predicted by the chosen repetition. $b$ can be a number chosen by us. Once we select the top-$b$ buckets, for each individual instance, the final scores of only those classes are computed that belong to top-$b$ buckets. Thus, the number of classes for which the final scores must be computed gets reduced to a number far less than the original number $K$.

For simplicity, let's refer to selecting the top-$b$ buckets as the stage 1 and calculating the final scores for classes in top-$b$ buckets as the stage 2. Also, the classifier used in stage 1 will be referred as the top-level MACH classifier and the classifiers used in stage 2 will be referred as the bottom-level MACH classifier.Therefore, we can say that the original MACH scheme comprises of $R$ bottom-level classifiers only.

• **Hypothesis 1**: In the naive approach, classes are randomly hashed into buckets based on universal hash functions, the classes hashed into similar buckets might not have similar contexts in which they appear. For example, consider a scenario where the number of buckets or meta-classes is 2 ($B = 2$) and the possible classes/labels are "pen", "stationary", "pencil", "cats" and "dogs". There's a possibility that "pen","pencil" and "dogs" get hashed into one bucket and the remaining labels get hashed into the other bucket. Such randomness in forming meta-classes might result into high error propagation from stage 1 to stage 2 when we try to use the naive approach, resulting into considerable performance degradation compared to the original MACH scheme.

• **Hypothesis 2**: Assuming that the *Hypothesis 1* is correct. Then high error propagation in naive approach can be mitigated by making bucket aggregations or meta-class formulations contextually meaningful with respect to data. This modification is proposed only for the repetition that is being used for stage 1 of the naive approach. Such an aggregation scheme would force labels occurring in similar contexts to occur in same meta-classes. Consider the example used in hypothesis 1, a contextual aggregation of classes would appear something like "pen","stationary" and "pencil" in one bucket and the remaining classes in the other bucket. This should result into lower error propagation from stage 1 to stage 2 of the proposed naive approach and thus, reducing the performance degradation with respect to the original MACH scheme.

From this point onward, the first approach to improving MACH's limitation would be referred as the Naive version of MACH or N-MACH and, the second and more promising approach would be referred to as Clustering enhanced MACH or C-MACH (because clustering has been used in the second approach which shall be discussed in section 3.4)

## 3.4 Clustering Enhanced Merged Average Classifier via Hashing

According to Hypothesis 2, meaningful aggregations for formulation of buckets or meta-classes should be done at stage 1. These meta-classes are then used by the top-level classifier as the ground truth. In our proposed method we make use of clustering to form these contextually aware meta-classes. And since, we tend to use clustering in the new scheme, we call this variant of MACH as *Clustering enhanced Merged Classifier via Hashing* or simply, *C-MACH*. In order to generate contextually aware meta-classes for the top-level classifier, two additional steps have to be undertaken before all the classifiers can be trained: Label Embedding Generation and Clustering.

### 3.4.1 Label Embedding Generation

In order to meaningfully aggregate similar labels in the same meta-classes, we require a way to represent labels such that they convey information about their context. But, this is not something that can be done straight away, whether a label is active or not for a data point, only conveys the ground truth information and nothing about the context in which labels are being used. In order to tackle this issue, we decided to use label embeddings. Label embeddings are simply a vector representation of the respective labels. These vectors comprise of real number values. For similar labels, label embeddings would also be similar. Based on the work of [8], we decided to implement three types of label embeddings: *Text-Embedding*, *Tf-idf Embeddings* and *Neural-Embeddings* embeddings.

- **Text-Embedding**: In this label embedding scheme, embeddings for raw text labels are obtained by using hidden embeddings of a pretrained *XLNet* [40] model. The version[1] of XLNet was pretrained on data which comprised of cleaning English text.

[1]The pretrained model was obtained from Transformers library. Refer `https://huggingface.co/transformers/pretrained_models.html` for more details

1024 is the dimensionality of the each embedding vector. XLNet is a state-of-the-art bi-directional auto regressive language model which is based on the already existing BERT [13] architecture. In circumstances where a raw label text comprises of more than one words or tokens, we simply take an average of the XLNet embeddings for all the constituent tokens of the label text.

$$\Phi_{text-embedding}(y) = \frac{1}{|text(y)|} \sum_{token \in text(y)} \phi_{XLNet-Pretrained}(token) \tag{3.3}$$

In the equation 3.3 [8], $y$ represents the raw text label, $token$ represents a token in $y$ and $\phi_{XLNet-Pretrained}(token)$ represents the embedding obtained from the hidden embedding layer of the pretrained XLNet for some $token$ in $y$.

The text-embedding approach seems promising when it comes to generating label embeddings that can give information about the contexts in which a label is used. But, there's a drawback to this approach, we are relying heavily on just a few tokens to obtain contextual information about a label, this might introduce a lot of unwanted noise in the process [8]. The answer to this issue is the use of information contained in the input features of those data points for which a label is active when generating label embeddings.

- **Tf-idf Embedding**: Tf-idf [30] is used to represent the importance of a token for a document in a collection of documents. Tf-idf comprises of a product of two values, *term frequency* and *inverse document frequency*. Term frequency is used to calculate the importance of a token for a document. Term frequency is the number of times a token occurs in a document normalized by the total number of tokens in a document. Since, term frequency gives more weight to more frequently occurring tokens, there's a need to reduce the importance of such tokens which, in general, occur a lot in the entire collection of documents, e.g. "the" is very common term and might not contribute much in distinguishing one document from other. Inverse document frequency is used exactly to counter this short-coming of term frequency. Inverse document frequency is the log of the ratio of total number of documents to the number of documents in which a particular token occurs. Equation 3.4, 3.5 and 3.6 summarise the process to compute Tf-idf values.

$$Term\ Frequency = \frac{Number\ of\ times\ a\ token\ occurs\ in\ a\ document}{Total\ number\ of\ documents} \tag{3.4}$$

$$Inverse\,Document\,Frequency = log_e\left(\frac{Total\ number\ of documents}{Number\ of\ documents\ in\ which\ a\ particular\ token\ occurs}\right) \tag{3.5}$$

$$Tf-idf = Term\,Frequency * Inverse\,Document\,Frequency \tag{3.6}$$

In order to compute tf-idf embedding for some label $l$, normalized sum of tf-idf values of all those data points for which label $l$ is active is calculated. Equation 3.7 [8] represents this calculation.

$$\Phi_{tf-idf\ embedding}(y) = \frac{emb_l}{||emb_l||},\ emb_l = \sum_{i:y_{il}=1} \phi_{tf-idf}(x_i),\ l = 1, ...., K \tag{3.7}$$

- **Neural-embedding**: This method of generating label embeddings uses hidden embeddings from the XLNet model which was fine-tuned on the cluster prediction task from the research carried out in [8]. Unlike *text-embedding* , embeddings for input tokens or features are computed rather than for the raw text labels. Just like the *tfidf-embedding*, normalized sum of embeddings (from the fine-tuned XLNet) of all those data points for which the label $l$ is active is calculated. Equation 3.8 [8] summarises the whole process of computing neural embeddings.

$$\Phi_{neural\ embedding}(y) = \frac{emb_l}{||emb_l||},\ emb_l = \sum_{i:y_{il}=1} \phi_{XLNet-Finetuned}(x_i),\ l = 1, ...., K \tag{3.8}$$

### 3.4.2 Clustering

Clustering is the process that groups similar data points together, in our case similar labels. Each cluster has an associated centroid which represents the position of a cluster in a hypersphere of $e$ dimensions. Any clustering mechanism follows the general structure of algorithm 1.

In our case, we have used spherical k-means algorithm for clustering, which uses cos dissimilarity (given by equation 3.9) as a measure of distance. Cosine dissimilarity originates from cosine similarity which measures the cosine of angle between two non zero vectors. A value of cosine close to 1 represents high similarity (see figure 3.2). The value of cosine similarity lies between -1 and 1 therefore, for cosine dissimilarity, cosine similarity has to be subtracted from 1. The reason why we chose cosine based spherical k-means clustering method over the more common euclidean distance based k-means clustering method is because some of the documents or inputs might be too

```
Algorithm 1: A General Clustering Algorithm
Data: Data $D$ , where $D$ is a matrix of size $NE$ , $N$ is the number of data points and $E$ is the
    dimensionality of each data point
Input: Number of clusters/centroids $k$
Output: Datapoints with cluster memberships
1: Initialize k centroids randomly
2: Associate each data point in D with the nearest datapoint based on a distance/loss function.
3: Recalculate the position of centroids
4: Repeat Step2 and Step3 until there are no more changes in the membership of the data points
```

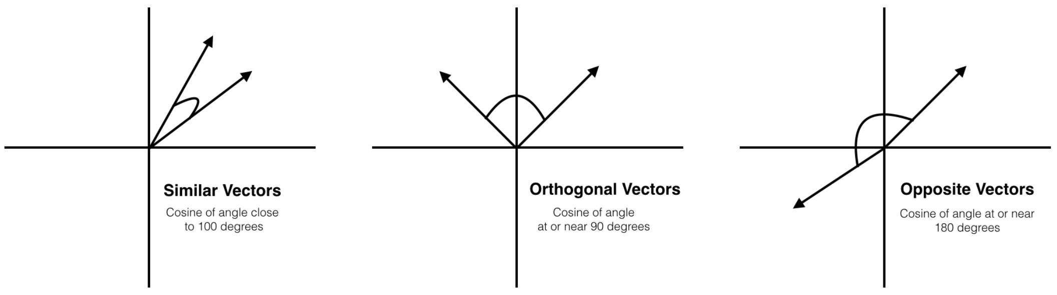

Similar Vectors
Cosine of angle close to 100 degrees
Orthogonal Vectors
Cosine of angle at or near 90 degrees
Opposite Vectors
Cosine of angle at or near 180 degrees

Figure 3.2: In the above feature, two dimensional non-zero vectors are considered. The angle subtended between any two vectors is representative of the similarity between two vectors. The value of subtended angle is incresing from left to right which means that cosine similarity is decreasing from left to right. Image taken from [33].

short (way more sparse than other documents) resulting into high euclidean distance for longer documents and small values for shorter documents. Cosine dissimilarity doesn't have this bias of euclidean distance because it doesn't consider magnitude of the features of the documents in calculating the similarity score (which is opposite of what is done in calculating euclidean distance ) but only the orientation, this removes the bias towards the length of the document.

$$d(x,x') = 1 - cos(x,x') = 1 - \frac{< x,x' >}{||x||||x'||} \tag{3.9}$$

Our clustering based approach is inspired by [8] in which authors have used balanced label clustering in their . In order to perform balanced clustering of labels, hierarchical balanced clustering was used. In this approach, a binary tree structure is used. Under balanced clustering, each parent node has two child nodes (which are formed by clustering) with equal number of instances in them. The process is recursive and continues until the number of instances in the leaf nodes reach below a certain set threshold count. By adjusting the threshold, the number of clusters can be changed. Though, unlike [8], we don't restrict ourselves two just balanced clustering but we also tried out unbalanced clustering to see what effect can this constraint bring in the performance of C-MACH. The motivation behind using unbalanced clustering comes from the work done by [19] according to which unbalanced clustering performed better than approaches similar to it that used balanced clustering. Unbalanced clustering process is straight forward and requires no use of any additional data structure like balanced clustering.

### 3.4.3 Partitioning & Training

**Algorithm 2:** Training algorithm for C-MACH

```
Data: $D = (X,Y) = (x_i, y_i)_{i=1}^{N}, x_i \in \mathbb{R}^d, y_i \in \{1,2....,K\}$
Input: $B,R,L$ (Note: $L$ is a matrix that stores K Label Embeddings. $L$ has dimensions $K * E$,
       where $E$ is the dimension of each label embedding)
Output: $R+1$ trained classifiers
 1: Initialize $R$ hash functions $h_1, h_2, ..h_R$
 2: Initialize a dictionary $C$
 3: Initialize $M$ as an empty list
 4: Cluster the $K$ classes using spherical k-means based on $L$ and update $C$ such that, $C(Y)$ represents
    the cluster to which a class belongs
 5: for $i = 1,2,\ldots,R$ do
 6:     $Y_{h_i} \leftarrow h_i(Y)$
 7:     $M_i = Train\,Classifier(X, Y_{h_i})$
 8:     Append $M_i$ to $M$
 9: end for
10: $Y_C \leftarrow C(Y)$
11: $M_{R+1} = Train\,Classifier(X, Y_C)$
12: Append $M_{R+1}$ to $M$
13: Return $M$
```

Once the clusters (or in other words semantically enhanced meta-classes) are obtained, we partition the large classification problem into smaller sub-problems using the same scheme as MACH (refer section 3.2.1). In C-MACH, there are $R$ bottom-level classifiers and one top-level classifier. Partitioning is followed by the process of training the top-level and the base-level classifiers in an embarrassingly parallel fashion. The architecture of the top-level classifier is kept constant as the MACH's bottom-level classifiers. In the original version of MACH, its classifiers were trained in parallel which is now still the case for C-MACH. The only difference now is that one of the $R+1$ classifiers, let's say the $R^{th}$ classifier, is now using semantically enhanced ground truth (meta-classes) for supervised training. The choice of loss function still remains the same for both the categories of classifiers, top-level and bottom-level. Therefore, training process remains the same. But what changes is the process of prediction which has been dealt next. The entire partitioning and training process is summarised by algorithm 2.

### 3.4.4 Prediction & the top-k result

Prediction stage of C-MACH is slightly different from the one that MACH uses. MACH uses a flat approach, i.e. it directly combines the results from all the $R$ classifiers, where as C-MACH uses a combination of a hierarchical and flat approach during prediction. C-MACH, as mentioned earlier, runs a forward propagation pass through the top-level classifier to compute the scores for all the possible semantically enhanced meta-classes (label clusters). It then performs an argmax or argsort on the label cluster scores depending on whether the dataset used is multi-class or multi-label, respectively. Then it uses the scores to select only the top-b clusters (in the case of multi-class $b=1$). Once the top-b clusters are obtained, all the classes that belong in the top-b clusters are passed on to base level classifiers using a dictionary that maps clusters-to-labels. This is followed by calculation of final scores for the classes in the top-b clusters, for which $R$ base-level classifiers were used with the same unbiased estimator as the original MACH.

Once the final scores are obtained for the classes in the top-b clusters, argmax or argsort operations are performed on only the labels for which the final scores are calculated and not for the entire pool of labels. For a clearer understanding, algorithm 3 can be referred.

**Algorithm 3:** Prediction algorithm for C-MACH

```
Data: $D = (X,Y) = (x_i, y_i)_{i=1}^{N}, x_i \in \mathbb{R}^d, y_i \in \{1,2....,K\}$
Input: $M = M_1, M_2, ..., M_R$ and $Q$
Output: $N$ predicted labels
 1: Load $R$ hash functions $h_1, h_2, ..h_R$ used in training
 2: Create a dictionary $inv_C$ such that it maps to all the classes in a cluster.
 3: Initialize $P$ as an empty list
 4: Initialize $G$ as a matrix with dimentionality $NK$
 5: for $i = 1, 2, \ldots, R$ do
 6:     $P_i = getProbability(X, M_i)$
 7:     Append $P_i$ to $P$
 8: end for
 9: $P_{R+1} = getProbability(X, Q)$
10: Append $P_{R+1}$ to $P$
11: for $i = 1, 2, \ldots, N$ do
12:     $cluster = argmax(P_{R+1}[i,:], axis = 1)$
13:     for $j$ in $inv_C(cluster)$ do
14:         $G[i,j] = (\sum_{r=1}^{R} P_r[i, h_r(j)])/(R)$
15:     end for
16: end for
17: Return $argmax(G, axis = 1)$
```

# Chapter 4

# Experiments and Evaluation

## 4.1 Software and Hardware

In our research, a lot of specialised tools and softwares were used. All the code for our research has been written in Python (3.5.6) [1]. Tensorflow [1], a library for implementing neural networks in Python has played a major role in implementation of our project. Scikit-Learn [26] (for mathematical operations), Scipy [36] (for handling sparse matrices) and Transformers [38] (for obtaining pre-trained embeddings) were also some of the important Python libraries for this project. In addition to Python libraries, parts of implementation of [8] on Github [2] were used in out project.

In terms of hardware, a system with 16GB memory, Intel Core i7 ($7^{th}$ generation) processor and a Nvidia GTX1050ti graphics card with display memory of 4 GB was used for implementation of our project. Compared to the CPU which has 4 cores , graphics card has 768 cores which makes training neural networks faster by using batching which leads to parallel computation of losses for instances in a batch. Graphics card can also be used to train several models together in parallel subject to the fact that there is enough display memory available in the graphics card.

[1]`https://www.python.org/`

[2]`https://github.com/OctoberChang/X-Transformer`

## 4.2 Datasets

### 4.2.1 Eurlex-4K

Eurlex-4K [24] is a multi-label dataset. It is a collection of documents that deal with European Union Law. These documents are related to different aspects of European Union Law, e.g. treaties, legislative proposals, etc. Some examples of the different categories in which these documents are classified into are: “unemployment insurance”, “treaty on European union”,“supplies contract”, “free-trade agreement”, etc. The total number of such categories is 3956. The documents are represented as bag of words features. Table 4.1 summarises some important numbers about Eurlex-4K.

| Property | Values |
|---|---|
| Number of train samples | 15449 |
| Number of test samples | 3865 |
| Dimensionality of features | 186104 |
| Cardinality of the Label Set | 3956 |
| Avg. number of samples per label | 25.971 |
| Avg. number of active labels per sample | 5.320 |
| Avg. document length | 270.723 |

Table 4.1: A summary of Eurlex-4K dataset

### 4.2.2 Wiki10-31K

This dataset is also multilabel. Wiki10-31K [44] comprises of documents taken from Wikipedia that existed on April 2009. All the documents were in English. These documents were enriched with social tags as part of a research that was carried out by authors of [44]. These social tags were obtained from a bookmarking site called Delicious. The total number of tags or labels are 30938. “sensors”, “revelation”,“outlaws” and “nitrogencycle” are some of the examples of the labels. Just like Eurlex-4K, the documents in Wiki10-31K are represented as Bag of Words features. Table 4.2 summarises some of the useful statistics about the dataset.

| Property | Values |
|---|---|
| Number of train samples | 14146 |
| Number of test samples | 6616 |
| Dimensionality of features | 101938 |
| Cardinality of the Label Set | 30938 |
| Avg. number of samples per label | 12.592 |
| Avg. number of active labels per sample | 18.764 |
| Avg. document length | 669.049 |

Table 4.2: A summary of Wiki10-31K dataset

## 4.3 Evaluation Metrics

### 4.3.1 Precision@k

Precision is one of the most common evaluation metrics used in multi-label classification. Precision is simply the ratio of all the relevant labels present in the prediction to all the labels in the prediction. Precision is essentially a measure for True Positives with respect to prediction. Consider a scenario where an instance can be assigned labels from only the set {pen, stationary, scissor, cup, writing, space-ship}. If an instance has true labels {pen, stationary, writing} and the predicted labels are {pen, cup, space-ship, stationary} then the precision is 0.5.

When Precision@k or P@k is computed, only the top-k predictions are considered for computation. For example, considering the previous example, if the order of predicted labels is in the decreasing order of confidence then Precision@3=0.33 because the only relevant item amongst {pen, cup and space-ship} is pen.

### 4.3.2 Recall@k

Just like Precision, Recall is also a very commonly used evaluation metric under a multi-label classification setting. Recall is the ratio of number of relevant labels in the prediction to the total number of true labels. Recall measures the sensitivity of the trained classifier. Considering the previous example, the recall is 0.67 as pen and stationary are the only relevant labels.

For Recall@k or R@k, only the top-k labels are considered. Based on our example, Recall@3 is 0.33 because pen is the only relevant item. In our experiments Recall@50 has been used.

### 4.3.3 Decoding Time

By decoding time, we refer to the combined time required to calculate the final label scores (time to forward propagation is not included in this) and to find the top-k labels. Since, reducing the decoding time is the main objective of our project, it only makes sense to use it as one of the evaluation metrics. Any improvement in the decoding time in our proposed versions of MACH directly associates with the success of this project and validation of our hypotheses.

## 4.4 Experiment & Results

In all our experiments, sparse format of the data was used, this reduces the amount of disk storage required for the data. Through out all the experiments that we have conducted, the architecture of all the neural networks were kept same as the ones used in [23] which had two hidden layers each with 500 nodes where, each of these hidden layers had Relu [25] activation functions. The original implementation of MACH [3] had used Adam [21] optimizer with initial learning rate of 0.001, $\beta 1$ value of 0.9, $\beta 2$ value of 0.999 and $\varepsilon$ value of $1e-07$. In our experiments too, we have used the same values for hyper parameters associated with the Adam optimizer. While performing the experiments we realized that 10 epochs were enough to produce results with performance close to the ones in the original MACH research. The last constraint, batch size was kept as 30 for all the experiments. A larger batch size would mean faster training as the number of parallel computations increase but with the system that we performed our experiments on, due to the limited memory, a maximum batch size of 30 could be used.

Our first step was to check whether our implementations of MACH, N-MACH and C-MACH have been correctly implemented. This was verified by comparing preliminary results from all the versions with a dummy classifier that predicts the most common label(s). This verification phase was the only exception where number of epochs was 4.

The next step was to train MACH for Eurlex-4K and Wiki10-31K to produce results comparable to the original MACH research. The bucket size of $\{64\ ,\ 128\}$ and

[3]`https://github.com/Tharun24/MACH`

| # of Repitition | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|
| Time | 20.531 | 22.372 | 22.713 | 24.511 | 27.63 | 33.752 |
| P@1 | 0.825 | 0.827 | 0.827 | 0.829 | 0.829 | 0.828 |
| P@3 | 0.71 | 0.713 | 0.714 | 0.718 | 0.717 | 0.718 |
| P@5 | 0.622 | 0.628 | 0.630 | 0.631 | 0.633 | 0.633 |
| Recall@50 | 0.369 | 0.410 | 0.435 | 0.451 | 0.463 | 0.471 |

Table 4.3: A summary of observation for MACH on Wiki10-31K dataset when the number of repetitions, $R$, was 6, 8, 10, 12, 14 and 16. The bucket size, $B$, used in this observation was 1024. Here, “Time” represents decoding time. Where as, P@k represents precision for top-k labels.

$\{512, 1024\}$ was used for Eurlex-4K and Wiki10-31K, respectively. By using just 16 repetitions, we were able to obtain a performance close to the result's reported in [23]. It has to be noted that when we compared the results form [23] and our implementation of MACH, we used results on the Wiki10-31K dataset (because results from [23] were observed for Wiki10-31K and not for Eurlex-4K). It was observed for both the datasets that with increase in $R$, after a certain point P@k stays almost constant but the Recall@50 kept increasing. This observation is evident from table 4.3 which represents results for MACH on Wiki10-31K dataset with $B$ value of 1024. It was also observed that with just 6 repetitions, acceptable results were achieved (refer table 4.3 for WIki10-31K) for both the datasets. The time required to train each repetition was 610 and 260 seconds for Eurlex-4K and Wiki10-31K dataset, respectively. But since all the repetitions can be trained in parallel, the total training time instead of being $16t$, was just $t$, where $t$ represents the total training time. The time required for forward propagation was 1.55239 and 0.756 milliseconds per sample respectively for Eurlex-4k and Wiki10-31K. On the other hand, time required for decoding was 0.514 and 4.254 milliseconds per sample for Eurlex-4K and Wiki10-31K dataset, respectively.

Following the experiments with MACH, N-MACH was trained and tested on the two datasets. As suggested earlier, N-MACH uses one of the 16 repetitions of MACH as top-level classifier and the others as base level classifiers. Since all the repetitions of the N-MACH are identical in terms of their structure and training procedure, any repetition could be chosen as the top-level classifier but, what if the one repetition that we choose for top-level classifier is the one that performs the worst amongst all other repetitions. Therefore, the results reported in table 4.4 for N-MACH are an

| | Metric | V-1 | V-2 | V-3 | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | tfidf | neu | temb | u-tfidf | u-neu | u-temb |
| Eurlex | Time | 2.057 | 1.044 | 1.138 | 1.154 | 1.162 | 1.475 | 1.566 | 1.762 |
| | P@1 | 0.803 | 0.801 | 0.801 | 0.802 | 0.801 | 0.802 | 0.801 | 0.8 |
| | P@3 | 0.673 | 0.668 | 0.672 | 0.672 | 0.667 | 0.672 | 0.672 | 0.665 |
| | P@5 | 0.565 | 0.553 | 0.562 | 0.564 | 0.551 | 0.565 | 0.564 | 0.554 |
| | R@50 | 0.818 | 0.655 | **0.796** | 0.788 | 0.662 | **0.814** | 0.811 | 0.708 |
| Wiki | Time | 28.154 | 2.814 | 2.566 | 2.548 | 2.544 | 4.038 | 4.457 | 4.509 |
| | P@1 | 0.828 | 0.827 | 0.8277 | 0.828 | 0.826 | 0.828 | 0.827 | 0.824 |
| | P@3 | 0.718 | 0.710 | 0.716 | 0.716 | 0.707 | 0.718 | 0.717 | 0.706 |
| | P@5 | 0.633 | 0.613 | 0.629 | 0.631 | 0.611 | 0.631 | 0.629 | 0.611 |
| | R@50 | 0.471 | 0.233 | **0.378** | 0.343 | 0.237 | **0.444** | 0.408 | 0.264 |

Table 4.4: A summary of observations made for MACH, N-MACH and C-MACH on Eurlex-4K and Wiki10-31K datasets. V-1, V-2 and V-3 represent MACH, N-MACH and C-MACH, respectively. "Time" refers to decoding time. tfidf, neural embedding (represented by "neu"), text-embedding (represented by "temb") represents the three types of embeddings when used for balanced clustering. A prefix "u-" before the embedding represents the case when embeddings were used for unbalanced clustering. All the results reported here are for bucket size, $B$, 128 and 1024 for Eurlex-4K and Wiki10-31K, respectively. The number of clusters in the top-level classifiers is same as the bucket-sizes of the corresponding base-level classifiers. 16 repetitions were used for V-1, 15 repetitions were used for N-MACH's base level classifiers and 15 repetitions were used for C-MACH's base level classifiers

average of 16 runs in which, each time a different repetition was chosen as the top-level classifier. From the results, it can be observed that the N-MACH's performance was very close to MACH's performance in terms of P@k but, in terms of Recall@50, N-MACH performed worse than MACH. The training and forward propagation time were the same as MACH but what was different was a much lower time requirement for decoding.

The next step was to experiment with C-MACH. The first thing that we did was the generation of the three types of label embeddings. The process of clustering adds extra time to the combined training process (training the repetitions + clustering), making C-MACH slower to train compared to MACH and N-MACH. Though, the time required

| | Metric | C-MACH | | | | | |
|---|---|---|---|---|---|---|---|
| | | tfidf | neural | temb | u-tfidf | u-neural | u-temb |
| Wiki10-31K | Time | 2.374 | 2.394 | 2.367 | 4.696 | 4.268 | 4.536 |
| | P@1 | 0.827 | 0.827 | 0.824 | 0.828 | 0.828 | 0.825 |
| | P@3 | 0.716 | 0.716 | 0.705 | 0.717 | 0.716 | 0.707 |
| | P@5 | 0.628 | 0.629 | 0.606 | 0.632 | 0.630 | 0.610 |
| | R@5 | 0.393 | 0.363 | 0.248 | 0.447 | 0.420 | 0.276 |

Table 4.5: A summary of observations made on C-MACH for different embedding types with the two different versions of label clustering. The bucket size, $B$, was 1024 where as the number of clusters was 512. These observations were made on Wiki10-31K dataset. Prefix "u-" represents the unbalanced label clustering based version of C-MACH.

for training the 15 usual repetitions and the clustering enhanced repetition is still the same as the previous two versions of MACH. Also, the time required for forward propagation through all the 16 models is still the same as MACH and N-MACH. C-MACH's time requirement for decoding was significantly lower than that of MACH. It is evident from table 4.4 that decoding time for N-MACH and C-MACH (balanced clustering version) were comparable. Between balanced and unbalanced clusters, the former performed better in terms of decoding time where as the later was better in terms of precision and recall. It is also worth mentioning that choice of $b$ for top-b clusters depends on the dataset, for Wiki10-31K it was 11 and for Eurlex-4K it was 3. A possible reason for a small value of $b$ to work for Eurlex might be because of a lower average number of labels per data point compared to Wiki which has a value close to 18.

Until now, all the experiments associated with C-MACH were performed with bucket size, $B$, values of 128 and 1024 for Eurlex-4K and Wiki10-31K, respectively, to see the performance when the cluster size is same as the bucket-size of the base-level classifiers/repetitions. The other version of C-MACH that we tested was the one in which cluster size was kept as 64 and 512 for the two datasets and the bucket size was kept higher, 128 and 1024 for the two datasets, respectively. Results for Wiki10-31K dataset for this version of C-MACH have been presented in table 4.5. The later version proves that, without any significant loss in performance, the number of clusters can be kept smaller than the bucket size. Although, it isn't recommended to have the number of clusters greater than the bucket size because in such a scenario, even though all the classifiers (one top level and the base level classifiers) are trained in an embarrassingly parallel way, top level classifier would take longer time for training than the base level classifiers. The same happens during the prediction time.

# Chapter 5

# Discussion

From the initial experiments conducted on Wiki10-31K dataset by using MACH, it was observed that addition of more repetition to MACH doesn't really help in increasing the precision@k. This might be because of the fact that some class labels are always scored relatively higher than the others which results into them always being in the top-k class labels if the value of *k* is small, e.g.when *k* is 5. We verified this by taking a test instance at index 1300 of the Wiki10-31K dataset and then observed the difference that occurs in labels that are included in the pool of top-5 classes as we increase the number of repetitions. The observations are summarised by table 5.1. From figure 5.1, it is evident that exactly three out five labels are always correct with respect to the example even though the number of repetitions is increased. This results into three labels: 22878, 25468 and 30134 always being predicted correctly. We can also observe that third highest scored label is a false positive which is always predicted. This is possibly because of the similarity between "wikipedia" and "wikipedi.us" which are represented by label IDs 30071 and 30134, respectively. Precision is too much dependent on how the classes are arranged on the basis of their class scores. It is also observed from table 4.3 that Recall@50, unlike Precision@k, keeps increasing with increase in number of repetitions. This implies that MACH is able to cover more true positives with increase in number of repetitions but the order of label scores might not be right with respect to the true labels.

From our experiments with N-MACH, we observe that N-MACH was able to successfully achieve the task of reducing decoding time. With N-MACH, decoding time was 1.97x and 10x faster for Eurlex-4K and Wiki10-31K datasets, respectively, with respect to that of MACH. We also observe that Precison@K for N-MACH is very close to that of MACH but, the case is entirely different for Recall@50. This might

| # of Repetitions | Top-5 Predictions in Increasing Order of their Scores |
|---|---|
| 6 | [ 2027 22878 30071 25468 30134] |
| 8 | [ 2027 22878 30071 25468 30134] |
| 10 | [23249 22878 30071 25468 30134] |
| 12 | [23249 22878 30071 25468 30134] |
| 14 | [23249 22878 30071 25468 30134] |
| 16 | [23249 22878 30071 25468 30134] |

Table 5.1: A summary of top-5 labels predicted by MACH for a test set instance (indexed at 1301; considering index starts at 1) of Wiki1031K. The bucket size, $B$, used in this observation was 1024. Out of all the labels predicted, the true labels are: 22878, 25468 and 30134 which represent the words *politicians*, *securities* and *wikipedi.us*, respectively. 30071 which is a false positive represents the word *wikepedia*

.

be because of the same reason because of which MACH's Precision@k stays constant even with increase in number of repetitions, i.e. some classes labels have a tendency to be ranked higher as compared to other class labels. The class labels that have a significantly higher class score compared to other other class labels, might be easily filtered through the top-level classifier of N-MACH and might enter into the base-level classifiers for score computation. Because of this, most of the class labels with low meta-class scores aren't able to reach the base-level classifiers, resulting into significantly lower Recall@50 value of N-MACH compared to MACH. This verifies our *Hypothesis 1*.

As suggested earlier in *Hypothesis 2* that if meta-classes are enhanced with label information there might not be a big drop in performance when pre-filtering class labels through the top-level classifier. This is exactly what was verified by observations made on C-MACH's experiments. It is also worth mentioning that, tf-idf based word embeddings were the best compared to finetuned and pretrained XLNet embeddings. From table 4.4, it can be observed that unbalanced clustering version of C-MACH performs better than the balanced one, but it comes at the expense of some extra decoding time. One can easily notice from table 4.4 that the version of C-MACH which uses balanced clustering and XLNet's pretrained embeddings performs as worse as N-MACH. Also, it is worth mentioning that C-MACH's performance gain with respect to decoding is almost as same as that of N-MACH but without any significant reduction in Recall@50.

# Chapter 6

# Conclusion and Future Work

In this project, we have proposed two versions of Merged-Average Classifier via Hashing or MACH which reduce the combined time required to compute class scores and to find the top-k classes (which we refer to as decoding time in this document). The two proposed methods work by pre-filtering the possible candidate classes before they are passed onto MACH for class score computation. This results into a significantly lower amount of computation compared to the original version of MACH. The two proposed methods were called N-MACH and C-MACH. N-MACH or Naive MACH uses the existing structure of MACH with only a slight modification. N-MACH uses one of the repetitions as a top-level classifier for pre-filtering the classes. On the other hand, C-MACH or Clustering enhanced MACH uses a top-level classifier whose label set (meta-classes) was enhanced by means of clustering labels based on their different types of embeddings, namely : tf-idf based embedding, pretrained XLNet's embeddings and finetuned XLNet's embeddings. It was observed that by the use of N-MACH and C-MACH an approximate reduction $2X$ and $10X$ was achieved on Eurlex-4k and Wiki10-31K datasets, respectively, with respect to decoding time. Though, it was the C-MACH which performed better due to its better Recall@50 compared to N-MACH.

Even though we were able to achieve the desired goal set aside in this project, there is a huge scope of improvement which can be done in future. For instance, the hyper parameter settings for neural networks used in C-MACH and N-MACH have been kept same as the one used in the original MACH implementation [23]. This can be experimented with in future. Another possible extension to our work can be investigation of performance of N-MACH and C-MACH with respect to tail labels and possible ways to improve it. In our project, all the experiments have been conducted on multi-label datasets. A further study can be carried out to observe how does N-MACH and C-MACH perform on multi-class datasets.

# Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Mohamed Aly. Survey on multiclass classification methods. *Neural Netw*, 19:1–9, 2005.

[3] Rohit Babbar, Ioannis Partalas, Eric Gaussier, and Massih R Amini. On flat versus hierarchical classification in large-scale taxonomies. In *Advances in neural information processing systems*, pages 1824–1832, 2013.

[4] Rohit Babbar and Bernhard Schölkopf. Dismec: Distributed sparse machines for extreme multi-label classification. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 721–729, 2017.

[5] Rohit Babbar and Bernhard Schölkopf. Data scarcity, robustness and extreme multi-label classification. *Machine Learning*, 108(8-9):1329–1351, 2019.

[6] Kush Bhatia, Himanshu Jain, Purushottam Kar, Manik Varma, and Prateek Jain. Sparse local embeddings for extreme multi-label classification. In *Advances in neural information processing systems*, pages 730–738, 2015.

[7] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[8] Wei-Cheng Chang, Hsiang-Fu Yu, Kai Zhong, Yiming Yang, and Inderjit S Dhillon. Taming pretrained transformers for extreme multi-label text classification.

[9] Yubo Chen, Liheng Xu, Kang Liu, Daojian Zeng, and Jun Zhao. Event extraction via dynamic multi-pooling convolutional neural networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 167–176, 2015.

[10] Anna E Choromanska and John Langford. Logarithmic time online multiclass prediction. In *Advances in Neural Information Processing Systems*, pages 55–63, 2015.

[11] Moustapha M Cisse, Nicolas Usunier, Thierry Artieres, and Patrick Gallinari. Robust bloom filters for large multilabel classification tasks. In *Advances in neural information processing systems*, pages 1851–1859, 2013.

[12] Leyang Cui and Yue Zhang. Hierarchically-refined label attention network for sequence labeling. *arXiv preprint arXiv:1908.08676*, 2019.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[14] Itay Evron, Edward Moroshko, and Koby Crammer. Efficient loss-based decoding on graphs for extreme classification. In *Advances in Neural Information Processing Systems*, pages 7233–7244, 2018.

[15] Daniel J Hsu, Sham M Kakade, John Langford, and Tong Zhang. Multi-label prediction via compressed sensing. In *Advances in neural information processing systems*, pages 772–780, 2009.

[16] Himanshu Jain, Venkatesh Balasubramanian, Bhanu Chunduri, and Manik Varma. Slice: Scalable linear extreme classifiers trained on 100 million labels for related searches. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 528–536, 2019.

[17] Himanshu Jain, Yashoteja Prabhu, and Manik Varma. Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 935–944, 2016.

[18] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[19] Sujay Khandagale, Han Xiao, and Rohit Babbar. Bonsai–diverse and shallow trees for extreme multi-label classification. *arXiv preprint arXiv:1904.08249*, 2019.

[20] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[21] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[22] Jingzhou Liu, Wei-Cheng Chang, Yuexin Wu, and Yiming Yang. Deep learning for extreme multi-label text classification. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 115–124, 2017.

[23] Tharun Kumar Reddy Medini, Qixuan Huang, Yiqiu Wang, Vijai Mohan, and Anshumali Shrivastava. Extreme classification in log memory using count-min sketch: A case study of amazon search with 50m products. In *Advances in Neural Information Processing Systems*, pages 13265–13275, 2019.

[24] Eneldo Loza Mencía and J. Fürnkranz. Efficient pairwise multilabel classification for large-scale problems in the legal domain. In *ECML/PKDD*, 2008.

[25] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] Yashoteja Prabhu, Anil Kag, Shrutendra Harsola, Rahul Agrawal, and Manik Varma. Parabel: Partitioned label trees for extreme classification with application to dynamic search advertising. In *Proceedings of the 2018 World Wide Web Conference*, pages 993–1002, 2018.

[28] Yashoteja Prabhu and Manik Varma. Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 263–272, 2014.

[29] Tomas Pranckevičius and Virginijus Marcinkevičius. Application of logistic regression with part-of-the-speech tagging for multi-class text classification. In *2016 IEEE 4th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, pages 1–5. IEEE, 2016.

[30] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142. New Jersey, USA, 2003.

[31] Najmeh Samadiani and Hamid Hassanpour. A neural network-based approach for recognizing multi-font printed english characters. *Journal of Electrical Systems and Information Technology*, 2(2):207–218, 2015.

[32] Yukihiro Tagami. Annexml: Approximate nearest neighbor search for extreme multi-label classification. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 455–464, 2017.

[33] Alex Tellez, Max Pumperla, and Michal Malohlava. *Mastering Machine Learning with Spark 2. x.* Packt Publishing Ltd, 2017.

[34] Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)*, 3(3):1–13, 2007.

[35] Shashanka Ubaru, Sanjeeb Dash, Arya Mazumdar, and Oktay Gunluk. Multilabel classification by hierarchical partitioning and data-dependent grouping. *arXiv preprint arXiv:2006.14084*, 2020.

[36] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser,

Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[37] Rahul Wadbude, Vivek Gupta, Piyush Rai, Nagarajan Natarajan, Harish Karnick, and Prateek Jain. Leveraging distributional semantics for multi-label learning. *arXiv preprint arXiv:1709.05976*, 2017.

[38] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771, 2019.

[39] Chang Xu, Dacheng Tao, and Chao Xu. Robust extreme multi-label learning. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1275–1284, 2016.

[40] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, pages 5753–5763, 2019.

[41] Ian EH Yen, Xiangru Huang, Wei Dai, Pradeep Ravikumar, Inderjit Dhillon, and Eric Xing. Ppdsparse: A parallel primal-dual sparse method for extreme classification. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 545–553, 2017.

[42] Ian En-Hsu Yen, Xiangru Huang, Pradeep Ravikumar, Kai Zhong, and Inderjit Dhillon. Pd-sparse: A primal and dual sparse approach to extreme multiclass and multilabel classification. In *International Conference on Machine Learning*, pages 3069–3077, 2016.

[43] Min-Ling Zhang, Yu-Kun Li, Xu-Ying Liu, and Xin Geng. Binary relevance for multi-label learning: an overview. *Frontiers of Computer Science*, 12(2):191–202, 2018.

[44] Arkaitz Zubiaga. Enhancing navigation on wikipedia with social tags. *arXiv preprint arXiv:1202.5469*, 2012.