COS 424 Final Project: Applying Online and Reinforcement Learning Techniques to Improve Cache Performance

Aninda Manocha
Princeton University
`amanocha@princeton.edu`

Abstract

As a result of the end of Moore's Law, specialized hardware has been developed to greatly improve computation performance and efficiency. However, accessing memory remains a bottleneck in many applications. Although designers have utilized caches for quicker memory access times to target the data supply bottlenecks, some applications continue to experience performance slowdowns. Graph algorithms in particular involve many irregular memory accesses due to the numerous unpredictable graph structures that exist. The relatively long latencies of such memory accesses motivate the exploration of using machine learning techniques to predict memory addresses that are not cached and therefore require longer access times. Thus, this project explores reinforcement learning with contextual bandits and online learning with classifiers to predict whether memory addresses will hit or miss in the cache given information about the address, program counter, and previous cache accesses. These predictions are then made ahead of time to enable data prefetching when necessary and improve cache performance. I find that contextual bandits tend to correctly predict more cache misses than the classifiers for a benchmark with random and irregular memory accesses, while the classifiers perform better on a graph algorithm. With these models, I achieve cache performance speedups for the two programs of up to 9.69x and 1.45x respectively.

1 Introduction

With the end of Moore's Law and the widespread use of increasingly complex applications, designers of modern computing systems have targeted high performance and aggressive computing goals through efficient and specialized hardware. In such systems, memory operations have grown orders of magnitude slower than computation, necessitating techniques to address data supply bottlenecks. As a result, local memory hierarchies called caches have been employed in processors to store frequently accessed memory addresses close to where the computation takes place. Caches in turn reduce the latencies of accessing the main large memory storage present in computing systems. In parallelizable applications, memory accesses are often regular and predictable, and are favored by caches that can exploit the temporal and spatial locality in these workloads. However, other applications, especially graph algorithms, are notorious for irregular memory accesses that can generate several cache misses. Thus, predicting such cache misses in advance can be beneficial for performance because data can be prefetched from main memory so that the time needed to retrieve this data is minimized. Given the memory trace of a program that contains addresses and program counters of the corresponding memory instructions, this project aims to improve cache performance by predicting cache accesses. The memory trace is first preprocessed with feature engineering applied to the addresses for a larger feature space. I apply reinforcement learning with six contextual bandits models and online learning with four classifiers. The models are then employed to make predictions far enough in advance that prefetched data is available for addresses predicted to not exist in the cache. Lastly, I compare the models' abilities to correctly predict cache misses and their effects on cache performance. This paper is organized as follows: Section 2 describes related research, Section 3 details the methodology of the project, Section 4 presents key prediction and performance results, and Section 5 summarizes and concludes this work.

2 Related Work

Machine learning has been applied to computer architecture research in a variety of ways. Jimenez et al. used the perceptron model to replace the traditional two-bit counters commonly used in branch prediction, allowing for dynamic prediction [7]. Similarly, this project explores dynamic prediction, but in the context of predicting cache accesses. Hashemi et al. employed deep neural networks for hardware prefetching as an improved alternative to standard stream and correlation prefetchers [6]. Deep neural networks have a large training overhead, which motivates the exploration of more lightweight models for prefetching. However, the work's data preprocessing techniques inspired some of the techniques applied in this project (to be described below). Additionally, Sakr et al. applied a neural network based technique that is trained online to learn repetitive memory access patterns in parallel processing applications [10, 11]. To incorporate the predictions into a real time system, this project applies online learning as well, but again avoids neural networks and their training overheads. Lastly, Peled et al. explore inherent program semantics and develop a context-based memory prefetcher using a reinforcement learning model based on contextual bandits [9]. This project also explores the capabilities of contextual bandits, but applies feature engineering to the addresses rather than relying on machine and code attributes. Another method of improving cache performance is predicting re-references. For example, Wu et al. designed a history counter table as a memory signature-based cache hit predictor in order to predict when memory accesses will receive future cache hits and modify the cache replacement policy accordingly [13]. Machine learning could also be applied to this idea, which would be an interesting future research direction.

3 Methods

3.1 The Data

The data for this project are memory traces of program executions. A memory trace consists of the memory addresses the program accesses in addition to the corresponding program counters (pointers that indicates where the processor is in a sequence of program instructions). There can also be additional information present, such as the program instruction that required the memory access. More specifically, I focus on the memory accesses of two programs:

1. *Graph Projections*: a graph algorithm that takes a bipartite graph as input and relates nodes in one partition based on neighbors they share in the other partition
   - This algorithm requires a network to operate on. I run it on data gathered on crime in Moreno Valley: http://konect.uni-koblenz.de/downloads/tsv/moreno_crime.tar.bz2. For the purpose of this project, there are many other suitable networks, such as data collected from DBPedia and YouTube [8].
   - For more information on the algorithm and its purpose, visit https://en.wikipedia.org/wiki/Bipartite_network_projection.
   - As a graph algorithm, this program can generate many cache misses depending on the size of the input and the cache. The graph structure also affects the patterns of cache hits and misses.
2. *Random Pointer Chase*: a benchmark that contains many random memory accesses that are nested inside pointer chasing (referencing a reference, etc.)
   - Random memory accesses are extremely likely to generate cache misses because they do not take advantage of the temporal and spatial locality built into cache design.

I chose these two programs in order to compare my techniques with two extremely different memory access patterns. My research group has been working on a simulator for efficient accelerator design that is capable of running programs (with their inputs) and capturing all of the assembly instruction sequences and memory accesses involved. Thus, I used the simulator to create the memory traces. For this project, I created a cache simulator that can determine which accesses hit or miss in real time when the cache is invoked. The access latencies are described in Table 1.

Access Type	Latency (Cycles)
Check if cache contains address	1
Retrieve data from cache (due to cache hit)	1
Retrieve data from main memory (due to cache miss)	250

Table 1: Cache Simulator Access Times

3.2 Pre-processing the Data

Since the data consists of memory traces, there is no missing data and all features are numerical values. However, the feature space only contains the memory address and the program counter. Memory addresses are sparse relative to the size of the address space (2^{64} addresses in a 64-bit machine), so I create a larger and more informative feature space by applying feature engineering. In a processor, memory addresses are usually read in one at a time and the entire trace is not known beforehand. However, the architecture can be modified to allow a single pass through the memory trace as part of preprocessing. With this modification, I first normalize the addresses by identifying the smallest one in the trace and subtracting it from all of the addresses (so that the addresses start at 0 and their relative distances are more apparent). Since caches access data at the granularity of a block (several accesses grouped together), I then take each address and normalize it to align with the start of a cache block:

$$\bar{x} = int(\frac{x}{B}) * B$$

where x is the address, B is the cache block size (in bytes), and $\bar{x}$ is the normalized address. I then consider three forms of additional preprocessing to shrink the address space and increase the feature space:

1. *Address Clustering*: K-means clustering is applied to the normalized addresses to create clusters of addresses [1]. Addresses are then labeled based on the cluster they are assigned to and one-hot encoding is performed on the cluster number [2].
2. *Address Space Division*: The address space is evenly divided and each address is labeled based on the division it falls in. One-hot encoding is then performed on these division labels.
3. *Address Filtering by Frequency Threshold*: Only the addresses that are accessed frequently enough (at least T times, where T is the frequency threshold) are considered as labels. One-hot encoding is then performed on this set of addresses (addresses that don't appear frequently enough in the trace simply have a feature value of 0 across all of the encodings).

Although I purposely increase the size of the feature space, performing dimension reduction is a useful method of identifying the most impactful and/or meaningful features. Performing Principle Component Analysis is the last data preprocessing step that is optional. I automatically find the number of components that explain 95% of the total variance [5] and then use this number to perform PCA.

3.3 Models

3.3.1 Reinforcement Learning with Contextual Bandits

The multi-armed bandits scenario aims to allocate a fixed and limited set of resources between multiple competing choices (arms) that each have a stochastic reward. The objective is to maximize the expected reward. Contextual bandits adapt multi-armed bandits policies to a scenario where information about the state of the environment (context) is used to help make predictions. Essentially, they are multi-armed bandits with covariates [4].

I use an open-source Python package with various implementations of contextual bandits, all of which have binary rewards. Binary classification algorithms are utilized as black-box oracles. In my models, there are two arms that represent 1) a cache hit and 2) a cache miss and the reward is whether or not the cache access prediction was correct (1 or 0).

I employ six variants of contextual bandits that are grouped as follows:

- *Adaptive Greedy Exploration*: selecting an arm according to models when a reward with high certainty is expected, and choosing another arm at random when not
  - *Adaptive Greedy Threshold*: the estimated expected reward is above a certain set threshold
  - *Adaptive Greedy Percentile*: the threshold is a moving percentile of the predictions
  - *Adaptive Active Greedy*: active learning is performed on top of *Adaptive Greedy Percentile*
- *Thompson Sampling*: choosing an arm with a probability proportional to it being the best arm
  - *Bootstrapped Thompson Sampling*: Thompson Sampling is performed through fitting several models per class on bootstrapped samples

  - *Softmax Explorer*: probabilities are determined by a softmax transformation on the decision function scores
- *Upper-Confidence Bounds*: taking an upper bound on the reward predicted for an arm (also referred to as ”optimism in the face of uncertainty”)
  - *Bootstrapped Upper-Confidence Bounds*: the upper-confidence bound is determined by taking a percentile of the predictions from a set of classifiers that are fit with bootstrapped samples

They are used with the following parameters:

- Black-box oracle: M = *LogisticRegression (random_state=0, solver='lbfgs')*
- $\alpha = \beta = 5$ (so that the expectation $\frac{\alpha}{\alpha+\beta}$ is equal to 0.5.

1. *Adaptive Greedy Threshold (base_algorithm = M, nchoices = 2, decay_type = 'threshold')*
2. *Adaptive Greedy Percentile (base_algorithm = M, nchoices = 2, beta_prior = $Beta(\alpha, \beta)$, decay_type = 'percentile', decay = 0.9997)*
3. *Adaptive Active Greedy (base_algorithm = M, nchoices = 2, beta_prior = $Beta(\alpha, \beta)$, active_choice = 'weighted', decay_type = 'percentile', decay = 0.9997)*
4. *Bootstrapped Thompson Sampling (base_algorithm = M, nchoices = 2, beta_prior = $Beta(\alpha, \beta)$)*
5. *Bootstrapped Upper-Confidence Bounds (base_algorithm = M, nchoices = 2, beta_prior = $Beta(\alpha, \beta)$)*
6. *Softmax Explorer (base_algorithm = M, nchoices = 2, beta_prior = $Beta(\alpha, \beta)$)*

3.3.2 Online Learning with Classifiers

I use four classifiers from Python's SciKitLearn Library [3]. Unless specified, I use the default parameters that SciKitLearn sets for each classifier:

1. *Decision Tree*
2. *Random Forest (n_estimators = 20)*
3. *K-Nearest Neighbors (n_clusters = 3)*
4. *Multinomial Nave Bayes*

These classifiers are initially trained with the first 5 memory addresses and their cache result labels (obtained beforehand). Then, the classifiers are trained online: each address is read in one at a time, the model makes a prediction, the cache is invoked to determine the true result, and the true results are used to refit the model and continue sequential training. Essentially, the results of the testing data (always 1 sample) become the new training data and the training data grows in size over time.

3.4 Methodology and Evaluation

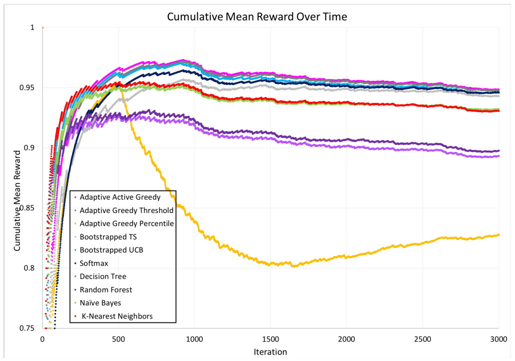
1. Perform data preprocessing. I mainly focus on address space division because it does not rely on reading the memory trace before applying the learning techniques. An exploration of the effect of address space division on the feature space size is detailed in Tables 2 and 3 in the Appendix.
2. Consider four main dataset variants:
   - *Random Pointer Chase*: divide address space into 2^{56} divisions, use large cache (32 KB) $\rightarrow$ very regular, frequent cache misses
   - *Graph Projections (Small)*: divide address space into 2^{58} divisions, use small cache (2 KB) $\rightarrow$ regular, occasional cache misses
   - *Graph Projections (Large)*: divide address space into 2^{58} divisions, use large cache (32 KB) $\rightarrow$ regular, very occasional cache misses
   - *Graph Projections (Small+Cluster+PCA)*: cluster address space into 2^{6} divisions, apply dimension reduction using PCA, use small cache (2 KB) $\rightarrow$ regular, occasional cache misses
3. Perform *cache access prediction* with all of the contextual bandits models and the classifiers and analyze their abilities to correctly predict cache misses.
4. Perform *prediction-based prefetching* with all of the contextual bandits models and the classifiers and analyze their abilities to improve cache performance. The prefetch distance is set to **250** because this is the latency of data retrieval due to a cache miss. By prefetching data this many cycles ahead for a correctly cache miss, the latency of data retrieval from main memory is effectively eliminated.
5. Evaluate the models' performance with respect to cache access prediction accuracy and cache performance (latency).

4 Results

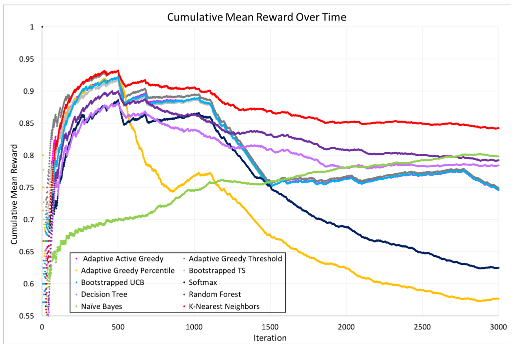
This project involves two main components: 1) accurately predict when an address will miss in the cache and 2) improve cache performance by prefetching data for addresses expected to miss in the cache. Below I describe the results and evaluations of these efforts.

4.1 Cache Access Prediction

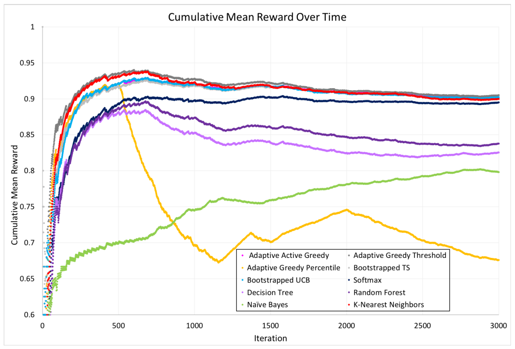
I first perform cache access prediction for each of the six contextual bandits models and each of the four classifiers. Since these models are continually refitted with the results of every new memory access, I measure the accuracies of these models over 3000 memory accesses. The accuracy of each prediction is equivalent to the reward, which is fed into the contextual bandits for reinforcement learning. These results are illustrated in Figure 1.



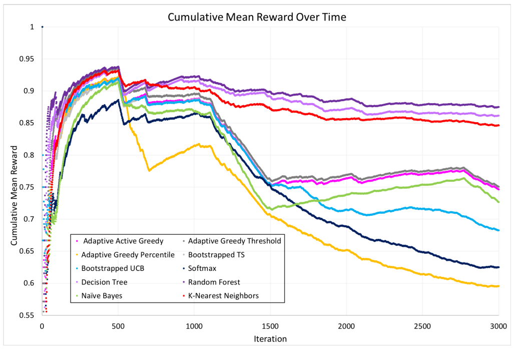
(a) *Random Pointer Chase*



(b) *Graph Projections (Small)*



(c) *Graph Projections (Large)*



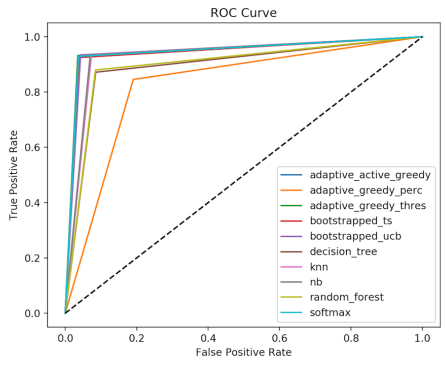
(d) *Graph Projections (Small+Cluster+PCA)*

Figure 1: Cumulative mean rewards over time for each of the four dataset variants.

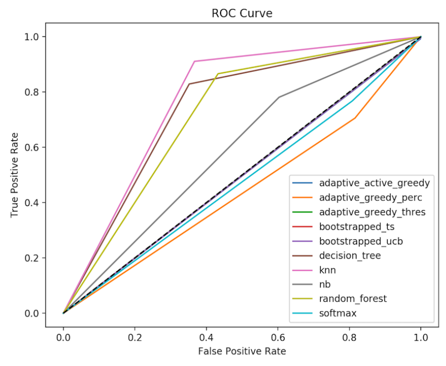
The results show that for *Random Pointer Chase* and *Graph Projections (Large)*, the contextual bandits generally yield greater accuracies than the classifiers. On the other hand, the classifiers tend to perform better for the *Graph Projections* variants with a small cache. Additionally, the cumulative rewards appear better when the data preprocessing involves clustering the addresses and performing dimension reduction. While one might expect that contextual bandits should always outperform the other models because they use context information to make predictions, the classifiers are effectively using context information by being refitted with the true cache results corresponding to each access.

Some of the models experience worsening accuracies over time because of unexpected changes in memory access patterns. For example, in *Graph Projections (Small)*, groups of cache misses appear randomly due to the structure of the graph input. Models whose accuracies improve over time (i.e. Nave Bayes) are likely learning the graph structures and forming predictions accordingly. Ideally, all models would ultimately learn the memory access patterns well to perfectly anticipate cache misses.

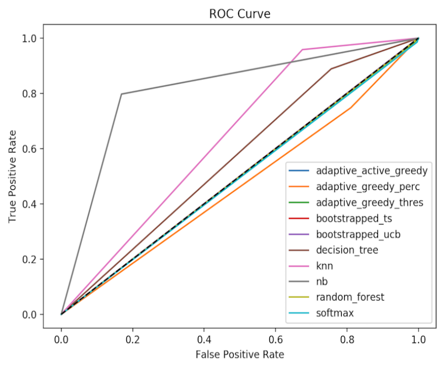
I dive further into the accuracies by looking at the false positive and false negative rates. The ROC curves are presented in Figure 2 and the evaluation metrics of interest (accuracy, predicted miss accuracy, false positive rate, false negative rate, and AUC) for each model are listed in Tables 4-7 in the Appendix.



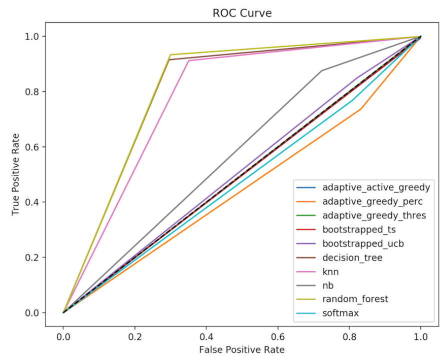
(a) ROC Curve for *Random Pointer Chase*



(b) ROC Curve for *Graph Projections (Small)*



(c) ROC Curve for *Graph Projections (Large)*



(d) ROC Curve for *Graph Projections (Small+Cluster+PCA)*

Figure 2: ROC curves for each of the four dataset variants.

Using *Random Pointer Chase* generates the best ROC curves, as the curves for all models are well above the identity function. This implies that the models do a good job of predicting cache accesses, as the false positive and false negative rates are fairly low for all models with the exception of *Adaptive Greedy Percentile*. The false positive rates are low for *Graph Projections (Large)* as well, but the false negative rates are quite high. For the two *Graph Projections* variants with a small cache, *Decision Tree*, *Random Forest*, and *K-Nearest Neighbors* have low false positive and false negative rates, while the other models have high false negative rates.

In order to ultimately improve cache performance, the models need to be able to correctly predict the occurrence of cache misses. Therefore, false positives (mispredicting a cache miss) are significantly more costly than false negatives (mispredicting a cache hit). Although mispredicting cache hits is not costly in terms of performance, I note that it is not ideal to unnecessarily fetch data and make the system do extra work. Therefore according to the metrics values, I anticipate that prediction-based prefetching using the models with the lowest false negative rates will improve cache performance the most. For example, *Random Forest* will yield one of the greatest cache performance improvements for *Graph Projections (Small+Cluster+PCA)*,

4.2 Prediction-Based Prefetching

I then perform prediction-based prefetching for each of the six contextual bandits models and each of the four classifiers. The idea of these experiments is that by forming cache access predictions ahead of time (predicting the result for an access x cycles in the future), data can be prefetched for addresses expected to need data retrieval from main memory. Thus, the amount of performance improvement likely depends on the prefetching distance because the number of cycles saved for a correctly predicted cache miss is $(251 - x)$, since 251 is the normal cache miss latency. This relationship is explored in Figure 3 specifically for *Random Pointer Chase*.

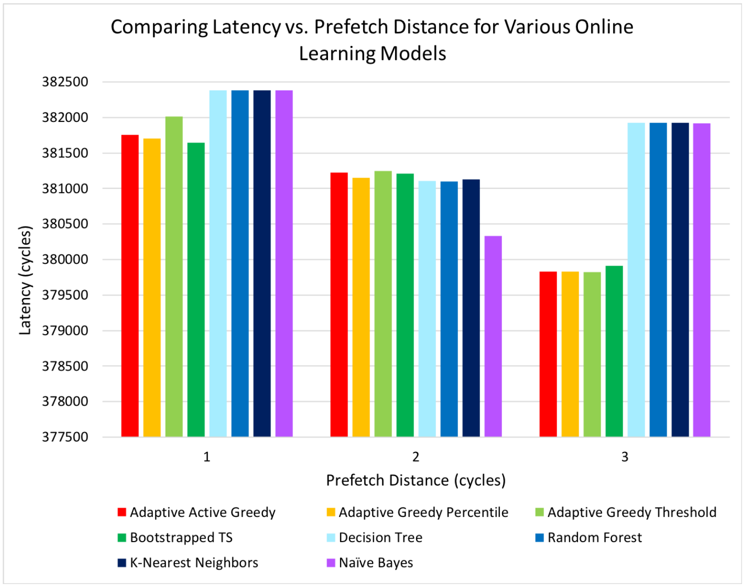


Figure 3: Comparisons of the effect of prefetch distance on cache performance for *Random Pointer Chase*.

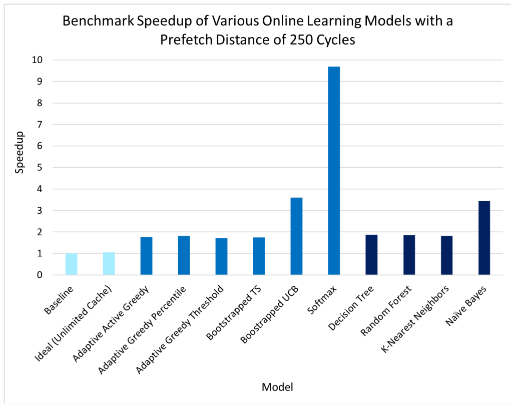
As the prefetch distance increases, the latency generally decreases for the models because increasing the prefetch distance increases the cache miss latency when a cache miss is correctly predicted. All four classifiers are an exception to the trend when the prefetch distance is 3 cycles, possibly because prefetching at this specific distance changes the memory access patterns (by introducing more cache hits) in a way that hurts the classifiers' abilities to predict cache misses. While I only present the effects of the prefetch distance for *Random Pointer Chase*, I anticipate a similar relationship between prefetch distance and cache performance for *Graph Projections*. Note: in a typical 5-stage processor pipeline [12], the decode stage (where memory addresses are determined) is 2 cycles before the memory stage, but the number of pipeline stages varies between different architectures.)

I then experiment with a prefetch distance of 250 (the latency of data retrieval from main memory) so that the long latency of cache misses is effectively removed when they are correctly predicted. Figure 4 highlight the cache performance improvements for each of the dataset variants and models when this prefetch distance is set. The models are also compared to an ideal cache (one with unlimited space that would reduce the frequency of cache misses).

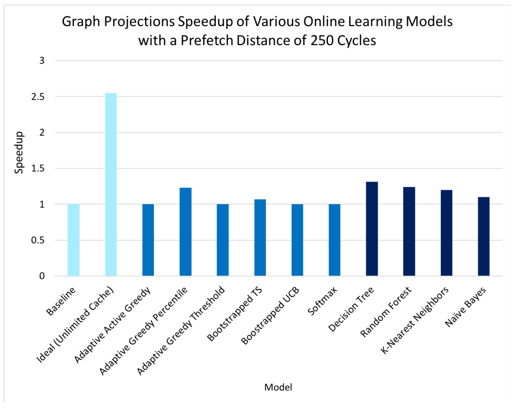
For *Random Pointer Chase*, all ten models produce a performance speedup ranging from 1.71x to 9.69x (averaging 2.93x) and well outperform the ideal cache. The contextual bandits tend to generate larger speedups, with *Softmax* doing exceptionally well. However, when I look further into why this is the case, I notice that with a prefetch distance of 250, the model predicts too many cache misses and prefetches very aggressively. While this eliminates cache miss mispredictions and is beneficial for cache performance, the architecture prefetches way more data than it needs to, which can be costly in terms of other metrics that are not emphasized in this project (i.e. energy).

For *Graph Projections (Large)*, the classifiers tend to perform better and all models average a 1.09x speedup. Comparing *Graph Projections (Small)* (average speedup of 1.12x) to *Graph Projections (Small+PCA+Cluster)* (average speedup of 1.13x), the differences in cache performance speedups are negligible for most models. Thus, performing address clustering and dimension reduction may have been slightly beneficial, but not significantly.

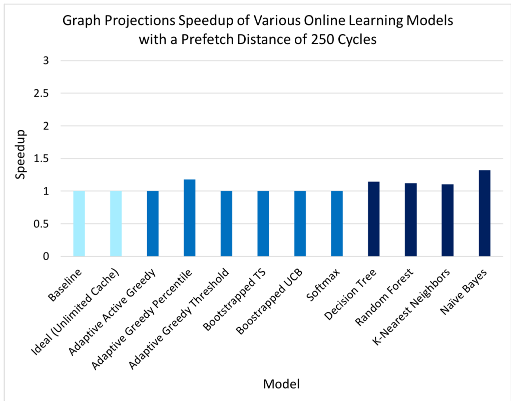
I originally expected the *Adaptive Greedy* contextual bandits to outperform the other models as David Cortes (the implementation creator) observed a similar observation [4]. The classifiers were intended to be baseline models for comparison, but I was pleasantly surprised by their abilities to predict cache misses and improve cache performance. This is good because the classifiers are more lightweight relative to many machine learning models used for prefetching and they operate quicker, allowing them to be more suitable for online learning in real-time systems.



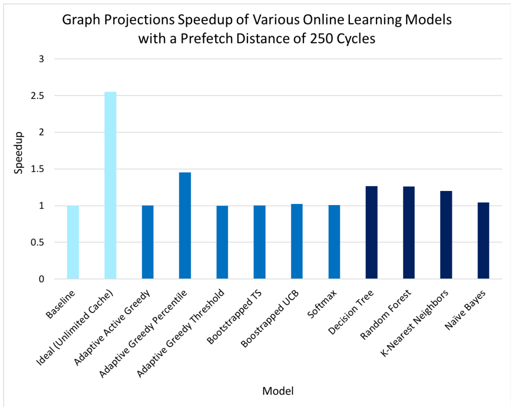
(a) *Random Pointer Chase*



(b) *Graph Projections (Small)*



(c) *Graph Projections (Large)*



(d) *Graph Projections (Small+Cluster+PCA)*

Figure 4: Cache performance speedups with each of the four dataset variants. (Light blue = no models, blue = contextual bandits, dark blue = classifiers)

5 Discussion and Conclusion

The main objective of this project was to utilize machine learning techniques, specifically reinforcement and online learning, to predict cache misses and ultimately improve cache performance with data prefetching. With *Random Pointer Chase*, many of the models (especially contextual bandits) were able to correctly predict cache misses and prefetch data at a distance of 250 cycles to generate performance speedups ranging from 1.71x to 9.69x. With *Graph Projections*, many models (some contextual bandits and all classifiers) were able to achieve performance speedups ranging from 1.01x to 1.45x. The models did a better job of predicting regular random memory accesses and there was a strong correlation between a model's false negative rate and its cache performance speedup. These results are informative for designing future architectures with prediction-based prefetching, as there are numerous extensions to this project that can refine and improve cache performance.

To begin with, I could use more programs (especially other graph algorithms) with varying memory access patterns since the dataset greatly impacts prediction and cache performance. I could also use more than 3000 accesses in each trace to see how the prediction accuracies and performance speedups scale. Next, since I mainly focus on address space division, I could explore and compare all three preprocessing techniques (with and without dimension reduction) in depth with regards to their effects on prediction, as one technique might generate a more informative feature space than the others. Since I utilize many different models and their default parameters, I could tune the hyperparameters (i.e. with cross validation) to improve their prediction abilities. Lastly, I could adjust my objective to predict re-reference patterns. I could extend the prediction of cache hits and misses to a prediction of memory accesses worth caching because they are likely to be re-referenced in the near future. Furthermore, the cache replacement policy could be modified based on the prediction of accesses that will not be re-referenced.

References

[1] Tola Alade. Tutorial: How to determine the optimal number of clusters for k-means clustering, 2018. Retrieved April 2019 from https://blog.cambridgespark.com/how-to-determine-the-optimal-number-of-clusters-for-k-means-clustering-14f27070

[2] Jason Brownlee. Why one-hot encode data in machine learning?, 2017. Retrieved April 2019 from https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/.

[3] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[4] David Cortes. Adapting multi-armed bandits policies to contextual bandits scenarios. *arXiv preprint arXiv:1811.04383*, 2018.

[5] Arthur Gonsales. An approach to choosing the number of components in a principal component analysis (pca), 2018. Retrieved April 2019 from https://towardsdatascience.com/an-approach-to-choosing-the-number-of-components-in-a-principal-component-analys

[6] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In *Proceedings of the 36th International Conference on Machine Learning*, 2018.

[7] Daniel A. Jimenez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA)*, pages 197–206, 2001.

[8] Jrme Kunegis. KONECT - the Koblenz Network Collection. In *Proceedings of the International Web Observatory Workshop*, pages 1343–1350, 2013.

[9] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 285–297, June 2015.

[10] Majd Sakr, C. Lee Giles, Steven P. Levitan, William Horne, Marco Maggini, and Donald M. Chiarulli. On-line prediction of multiprocessor memory access patterns. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1564–1569, 1996.

[11] Majd Sakr, Steven P. Levitan, Donald M. Chiarulli, William Horne, and C. Lee Giles. Predicting multiprocessor memory access patterns with learning models. In *Proceedings of the 36th International Conference on Machine Learning*, pages 305–312, 1997.

[12] Wikipedia. Classic risc pipeline, 2019. Retrieved May 2019 from https://en.wikipedia.org/wiki/Classic_RISC_pipeline#The_classic_five_stage_RISC_pipeline.

[13] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 430–441, New York, NY, USA, 2011. ACM.

Appendix

Number of Address Space Divisions	Number of Features	Number of Address Space Groups
2^0 ... 2^{46}	6	2
2^{45}	7	3
2^{46}	9	5
2^{47}	10	6
2^{48}	13	9
2^{49}	19	15
2^{50}	31	27
2^{51}	55	51
2^{52}	105	101
2^{53}	204	200
2^{54}	393	389
2^{55}	694	690
2^{56}	1006	1002
2^{57}	1255	1251
2^{58} ... 2^{64}	1427	1423

Table 2: Address Space Sizes for *Random Pointer Chase*

Number of Address Space Divisions	Number of Features	Number of Address Space Groups
2^0 ... 2^{17}	6	2
2^{18} ... 2^{24}	7	3
2^{25} ... 2^{43}	8	4
2^{44}, 2^{45}	9	5
2^{46}	10	6
2^{47}	13	9
2^{48}	17	13
2^{49}	26	22
2^{50}	45	41
2^{51}	83	79
2^{52}	158	154
2^{53}	291	287
2^{54}	454	450
2^{55}	627	623
2^{56}	798	794
2^{57}	967	963
2^{58} ... 2^{64}	1163	1159

Table 3: Address Space Sizes for *Graph Projections*

Model	Accuracy	Predicted Miss Accuracy	False Positive Rate	False Negative Rate	Area Under ROC Curve
Adaptive Active Greedy	0.948	0.932	0.037	0.065	0.948
Adaptive Greedy Percentile	0.827	0.846	0.186	0.158	0.828
Adaptive Greedy Threshold	0.948	0.930	0.037	0.067	0.948
Bootstrapped TS	0.943	0.925	0.041	0.072	0.943
Bootstrapped UCB	0.946	0.934	0.045	0.063	0.946
Softmax	0.946	0.929	0.039	0.068	0.946
Decision Tree	0.893	0.872	0.090	0.121	0.894
Random Forest	0.898	0.880	0.089	0.115	0.898
K-Nearest Neighbors	0.931	0.935	0.074	0.065	0.931
Nave Bayes	0.932	0.934	0.071	0.065	0.932

Table 4: AUC Values for *Random Pointer Chase*

Model	**Accuracy**	**Predicted Miss Accuracy**	**False Positive Rate**	**False Negative Rate**	**AUC**
Adaptive Active Greedy	0.746	0.992	0.249	0.947	0.497
Adaptive Greedy Percentile	0.576	0.706	0.276	0.829	0.445
Adaptive Greedy Threshold	0.749	0.996	0.248	0.900	0.499
Bootstrapped TS	0.746	0.990	0.249	0.852	0.498
Bootstrapped UCB	0.747	0.992	0.249	0.900	0.498
Softmax	0.625	0.767	0.258	0.786	0.480
Decision Tree	0.784	0.829	0.123	0.445	0.748
Random Forest	0.792	0.866	0.141	0.417	0.717
K-Nearest Neighbors	0.842	0.911	0.117	0.299	0.772
Nave Bayes	0.686	0.781	0.203	0.626	0.589

Table 5: AUC Values for *Graph Projections (Small)*

Model	**Accuracy**	**Predicted Miss Accuracy**	**False Positive Rate**	**False Negative Rate**	**AUC**
Adaptive Active Greedy	0.747	0.992	0.249	0.944	0.497
Adaptive Greedy Percentile	0.595	0.736	0.271	0.836	0.452
Adaptive Greedy Threshold	0.751	0.997	0.248	0.750	0.500
Bootstrapped TS	0.746	0.991	0.249	0.870	0.498
Bootstrapped UCB	0.683	0.849	0.242	0.719	0.513
Softmax	0.875	0.933	0.097	0.225	0.480
Decision Tree	0.862	0.915	0.097	0.269	0.809
Random Forest	0.875	0.933	0.097	0.225	0.816
K-Nearest Neighbors	0.846	0.912	0.113	0.293	0.779
Nave Bayes	0.726	0.875	0.215	0.579	0.575

Table 6: AUC Values for *Graph Projections (Small+Cluster+PCA)*

Model	**Accuracy**	**Predicted Miss Accuracy**	**False Positive Rate**	**False Negative Rate**	**AUC**
Adaptive Active Greedy	0.889	0.996	0.108	0.947	0.499
Adaptive Greedy Percentile	0.699	0.748	0.096	0.932	0.469
Adaptive Greedy Threshold	0.905	0.997	0.093	1.00	0.499
Bootstrapped TS	0.902	0.993	0.092	0.900	0.500
Bootstrapped UCB	0.902	0.993	0.092	0.900	0.500
Softmax	0.895	0.986	0.093	0.975	0.495
Decision Tree	0.830	0.889	0.079	0.819	0.566
Random Forest	0.842	0.917	0.091	0.912	0.499
K-Nearest Neighbors	0.900	0.959	0.067	0.557	0.642
Nave Bayes	0.801	0.797	0.021	0.708	0.814

Table 7: AUC Values for *Graph Projections (Large)*