Casting Role Allocation through SAT solving and neural networks

12th November 2021

David Lang

A research project completed in partial fulfilment of a Masters of Engineering

Acknowledgements

I would like to thank the following people for their help throughout the project, without whom my project would not have been possible;

- Dr. Andrew Johnston, my supervisor, for his ideas and feedback throughout the project
- Dr. Robert Lang, for his years of work on the original Casting Expert software
- Cumberland Gang Show, for providing the historical casting data for use as a case study
- The CGS production team, specifically Kerrin Alamango, Linda Apthorpe, Lachlan Davies, Tristan Hornstra, Adele Lockman, Matt & Meagan Thompson, Luke Vella and Dr. Jeanie Wood, for their feedback and user acceptance testing
- Displayr, for use of their cloud-based data analysis software

I would also like to thank my family and friends for supporting me throughout my masters degree and my dear friend caffeine, which is of course the raw material that software engineers turn into computer code.

Abstract

Community and not-for-profit theatre is a common hobby for all ages, but casting people into roles appropriate for their ability level can be a challenging task, particularly when considering large groups and alternating casts of children.

This project sets out to evaluate the use of SAT solving and neural network based algorithms for the selection of cast and allocation of roles in theatrical productions, and incorporate the successful concepts into a working software solution.

In order to implement the required algorithms, the boolean satisfiability problem (SAT) and artificial neural networks (ANNs) have been researched. A critical review has also been completed of the software packages previously used by Cumberland Gang Show (CGS), a production company which is used as a case study.

The hypotheses of this research are that SAT solving algorithms will improve the balance of talent between alternative casts, and that machine learning neural network models will improve the accuracy of recommendations and spread of roles allocated to applicants, over the previously used heuristic algorithms.

Six SAT based algorithms for cast selection have been implemented and assessed, with five of them found to improve on the heuristic baseline, and four improving on the historical cast selection as performed manually by the production team. This proves that SAT solving algorithms will improve the balance of talent between alternative casts.

Three ANN based algorithms for role allocation have been implemented and assessed, with various parameters totalling 32 neural models. All of these were found to improve accuracy over the heuristic baseline, proving that neural networks will improve the accuracy of recommendations, however the spread of roles was found to be less, disproving that hypothesis. A thorough discussion is included on the reasons why this may have occurred, and future research which could be conducted in this area.

The completed software package, implementing the successful algorithms, has been named “CARMEN: Casting And Role Management Equality Network” and released open source, under the GPLv3 license. The user acceptance testing by five members of the CGS production team found it was faster, easier to use and more intelligent than the previous software, with all five recommending it for use by other production companies.

This deliverable, and the research that went into it, has improved the quality of the casting process of theatrical productions. It will first be used for a real world production by the Cumberland Gang Show in December 2021, as they start auditions for their July 2022 performance, and will help the volunteer

production team to share roles fairly amongst applicants, leaving them to focus on what they do best; developing the talent, creativity and passion for the arts of the youth (and youth-at-heart) of Sydney, Australia.

Table of Contents

List of Figures

List of Acronyms

ANN	Artificial neural network
CDCL	Conflict-driven clause learning algorithm
CE	Casting Expert (the previously used in-house software package used by CGS)
CGS	Cumberland Gang Show (the theatrical company used as a case study in this project)
CNF	Conjunctive normal form
CRUD	Create, read, update, delete operations
CSV	Comma separated values (file format)
DOS	Disk operating system (also called MS-DOS)
DPLL	Davis-Putnam-Logemann-Loveland algorithm
EF	Entity framework (specifically meaning EF Core in this project)
GPLv3	General public license v3.0 (an open source software license)
GNN	Graph neural network
GUI	Graphical user interface
IDE	Integrated development environment
JSON	JavaScript object notation (file format)
LTR	Learning to rank
ML	Machine learning
MLP	Multi-layer perceptron
MLR	Machine-learned ranking
MVP	Minimum viable product
ORM	Object-relational mapper
POCO	Plain old class object (also called plain old CLR object, or colloquially plain old C# object)
RDB	Relational database
RNN	Recurrent neural network
SAT	Boolean satisfiability problem (sometimes called propositional satisfiability problem)
SDK	Software development kit
SGD	Stochastic gradient descent
SLP	Single-layer perceptron
TSV	Tab separated values (file format)
UAT	User acceptance testing
WIN	Microsoft Windows operating system
WPF	Windows presentation foundation (GUI framework)

1. Introduction

Community and not-for-profit theatre is a common hobby for all ages, but especially in school age groups with the ever increasing popularity of glee clubs, dance ensembles and the NSW School Spectacular. Casting people into roles appropriate for their ability level has become a challenging task, particularly when considering large groups and alternating casts of children.

This project will investigate the use of machine learning and other advanced algorithms as prediction and balancing mechanisms for the allocation of roles within a theatrical production. The results of this investigation will guide the creation of an open source desktop application which can be used by theatrical groups to assist in the casting process, based on their own show structure, candidate ability measures and casting rules.

One such show which fits this description is the Cumberland Gang Show, which performs annually with a cast of 160+ people ranging from 10 to 30 years old. Generally speaking each person gets allocated 10 roles out of approximately 30 items, with each item having numerous roles which need to be filled based on ability. At a basic level roles are allocated based on weighted ability marks, but there are further casting challenges involved due to having 2 separate casts which alternate performance nights and the limited availability of microphones and other resources. One of the primary goals of this application will be to evenly distribute talent between the 2 casts, but on top of this there are personnel conditions such as keeping siblings in the same cast which must be adhered to. As such, the Cumberland Gang Show will make an excellent case study for this project.

The engineering problem to be solved is the combination of hard and soft rules, including how to structure this in a generic enough manner that the software is useful for a broad range of theatre groups rather than being written solely for one. In order to solve the mathematical balancing problem I have researched and implemented SAT solving algorithms and artificial neural networking approaches to create a guided semi-automatic process for casting which identifies the critical roles and leads the user to choose these first and continue accepting (or modifying) recommendations until all roles have been cast.

An ideal solution to this problem will be one that can automatically cast the whole show, if required, but is more often used interactively to make choices and fine tune in a logical manner that does not require re-visiting old decisions due to an undesirable outcome. Formally speaking, my research question will be to find out whether SAT solving and machine learning algorithms, such as neural networks, can improve the quality of an automated casting system.

I believe this project has the potential to improve the quality of, and greatly reduce the time required to complete, the casting process of medium-large theatrical productions, especially those which aim to develop talent, share roles, and are organised by volunteers with limited time.

2. Background

In order to properly investigate, assess and implement the required algorithms, a thorough understanding of the boolean satisfiability problem (SAT) and artificial neural networks (ANNs) will be required. A review of the literature on these topics is included in sections 2.1 and 2.2 respectively.

Following the theoretical background knowledge, a critical review of the past software packages which Cumberland Gang Show (CGS) has used for the casting process has been undertaken in section 2.3. This has then been combined with feedback from the CGS production team (Appendix A - Previous user feedback) to form a complete list of functional requirements (Appendix B - Functional requirements) of the new software to be created.

2.1 Boolean satisfiability problem

The boolean satisfiability problem (SAT) is a class of problems in computer science which requires determining if there is *any* solution to a specific boolean expression. That is, whether or not there is any combination of true/false values assigned to the variables $x_1 \ldots x_n$ such that the expression evaluates to true.

While it is trivial to verify that a given set of value assignments is a valid solution, finding a solution (or proving that it is unsolvable) is not. More specifically, it is possible to verify a solution in linear time ($O(n)$) with respect to the number of variables n, but the search for a solution cannot be completed in linear, or polynomial time (eg. $O(n^2)$) and instead takes exponential time (eg. $O(2^n)$). This property makes SAT part of the set of NP-complete problems, one of the most important and well-researched areas of computer science in the 20th century.

The most basic implementation for solving SAT problems is a brute force method, whereby every possible combination of true/false values are assigned to the variables, however as this will take $O(2^n)$ time, this quickly becomes impractical as n increases. Although all currently discovered algorithms for solving the SAT problem still take exponential time, the best algorithms are currently capable of solving generalised SAT problems in ~$O(1.3^n)$ time, which makes this a viable approach for many applications where brute force would not be.

2.1.1 Conjunctive normal form

To formally define the boolean satisfiability problem, boolean algebra must be considered.

In boolean algebra, a variable x_i can either take the value of exactly 0 or exactly 1.

$x_i \in \{0, 1\}$

A literal is either a positive literal, just a variable x_i, or a negative literal, its negation $x_i\neg$.

x_i	$x_i\neg$
0	1
1	0

The primary boolean operators are the conjunction operator ∧ (AND), and the disjunction operator ∨ (OR).

x_i	x_j	$x_i \wedge x_j$	$x_i \vee x_j$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

A boolean expression is any combination of literals, operators and parentheses, which are used to define the order of operations. For example:

$f(x) = ((x_1 \vee x_2) \wedge x_3) \vee ((x_1\neg \wedge x_2\neg) \vee (x_3\neg \wedge x_i))$

In the context of SAT solving, boolean expressions are normally expressed in conjunctive normal form (CNF), that is as an “AND” of “ORs”. For example:

$f(x) = (x_1 \vee x_2 \vee x_3) \wedge (x_4\neg \vee x_2\neg \vee x_5\neg) \wedge (x_3\neg \vee x_6 \vee x_4)$

More generally, each disjunction (OR) of literals is called a clause:

$c_j = l_{j1} \vee l_{j2} \vee \ldots \vee l_{jk}$

And the expression is defined as a conjunction (AND) of clauses:

$f(x) = c_1 \wedge c_2 \wedge \ldots \wedge c_j$

When combined, this describes a boolean function of n variables as being k-CNF:

$f(x_1,\ldots,x_n)_{CNF} = (l_{1,1} \vee l_{1,2} \vee \ldots \vee l_{1k}) \wedge (l_{2,1} \vee l_{2,2} \vee \ldots \vee l_{2k}) \wedge \ldots \wedge (l_{j1} \vee l_{j2} \vee \ldots \vee l_{jk})$

where:

$l_{jk} \in \{x_1,\ldots,x_n, x_1\neg,\ldots,x_n\neg\}$

with j clauses of up to k literals per clause.

All boolean expressions can be expressed in conjunctive normal form (CNF), and in fact it has been proven that any arbitrary boolean expression can be converted to CNF in polynomial time, with respect to the

number of variables n. This form is particularly useful for most SAT solving algorithms, for reasons that are explained in 2.1.3 DPLL-based algorithms below.

As a final note on normal forms, it's worth adding that I did find one exception to the use of CNF in my research of SAT solving algorithms, that of Budinich's approach using Clifford algebra. They state that while in general "expansion to DNF is a terrible algorithm for SAT", it does help provide meaning in Clifford algebra, however they conclude that it is "not yet clear whether (this) can produce a competitive algorithm" (Budinich, 2019), and as such I haven't discussed it any further in this study.

2.1.2 Theme and variations

As previously mentioned, SAT is a generalised class of problems, with many specific versions often used. A brief summary of the key specific problems of the SAT class is provided in Figure 1 (below).

2-SAT	Where the satisfiability problem can be reduced to CNF with at most 2 literals per clause
3-SAT	Where the satisfiability problem can be reduced to CNF with at most 3 literals per clause
1in3-SAT	Like 3-SAT, but where exactly 1 of the 3 literals in each clause is true (not 2 or 3)
MAX-SAT	Where the maximum number of satisfiable clauses is found, when not all are satisfiable simultaneously
Weighted MAX-SAT	Like MAX-SAT, but where each variable has a cost function associated, and the goal to find the minimum (or maximum) sum of costs
ALL-SAT	Where all solutions to the satisfiability problem are found, or the number of such solutions is found
SMT	Satisfiability Modulo Theories problems are an extension of SAT whereby there is extra or external logic which must be adhered to rather than purely being satisfied by meeting the boolean criteria only

Figure 1 - Summary of SAT-like problems

Each of these problems are classed as restricted-SAT problems, because they are part of the SAT problem class but contain restrictions on various parameters, except for SMT which is an extension to SAT. Algorithms can be designed or optimised to solve these specific restrictions faster than the unrestricted-SAT problem, however they mostly follow one of two main approaches (Moskewicz, 2001), see 2.1.3 DPLL-based algorithms and 2.1.4 Local search algorithms below.

2.1.3 DPLL-based algorithms

In the early 1960s, Davis/Putnam and later Davis/Logemann/Loveland proposed a complete search algorithm to solve the boolean satisfiability problem, later named the DPLL algorithm after its 4 creators. This algorithm was, and still is, a cornerstone of the SAT solving process, being the basis of almost all complete SAT solvers (Moskewicz, 2001).

The DPLL algorithm was a significant improvement on the previously best 2 approaches;

1. That of Hao Wang's program using "a formulation of quantification theory… involved exponentiation on the total number of truth-functional connectives", and
2. P. C. Gilmore's program using normal forms, which "involves exponentiation on the number of clauses".

These were of course an improvement on the truth table table method, which involved "exponentiation on the total number of variables, (but) both run into difficulty with some fairly simple examples" (Davis, 1962).

The DPLL approach is based on the following 3 key rules (Davis, 1960).

I. Rule for the Elimination of One-Literal Clauses
   A. If the formula, expressed in CNF, contains any clauses with only a single literal, then that literal must be true in order for the formula to be satisfiable.
      1. The single literal clause may be removed.
      2. Any clauses containing the same literal, may be removed.
      3. Any instances of the inverse literal can be removed from their clauses.
   B. If any clauses contain no literals at any time, the formula is unsatisfiable.
   C. If no clauses remain at any time, the formula is satisfiable.

II. Affirmative/Negative Rule
   A. If a variable only ever appears in the formula as a positive literal, it can be set to true and all clauses which contain it can be removed.
   B. Similarly if a variable only ever appears in the formula a negative literal, it can be set to false and all clauses which contain it can be removed.

III. Rule for Eliminating Atomic Formulas
   A. If any clauses contain a literal and its inverse, then the clause can be removed*
   B. Put the formula into the form $(A \vee p) \wedge (B \vee p\neg) \wedge R$, where A, B, R do not contain p, by grouping clauses containing p and $p\neg$, and factoring out respectively
   C. The formula can now be reduced to $(A \vee B) \wedge R$**

** Although this is not specifically mentioned by Davis et al., it logically follows that it must be part of this rule in order to be able to put the formula in the correct form for III-B.*

*** Davis 1962 states that III-C is more practically used for computation by branching the function and testing $A \wedge R$ and $B \wedge R$ separately, as if either one is true, the overall result is true.*

The pseudocode for the basic DPLL algorithm is shown in Figure 2 below.

A keen observer will note that the DPLL algorithm makes no statement of how the implementer should choose the branching literal, and as such this is where much of the last 50 years of improvement to this algorithm has been situated.

Input: *a list of clauses in CNF*	***Output:*** *true or false*

```
function IsSatisfiable(clauses)
    for each clause in clauses
        if clause contains only 1 literal then
            remove clause from clauses # Rule I-A-1
            remove any clause containing literal # Rule I-A-2
            remove inverse of literal from any clauses # Rule I-A-3
    for each unique literal in all clauses
        if literal only occurs positive or literal only occurs negative then
            remove all clauses containing literal # Rule II
    if clauses is empty then
        return TRUE # Rule I-C
    if any clause contains no literals then
        return FALSE # Rule I-B
    choose a branching literal L from the remaining literals
    return IsSatisfiable(clauses with single literal clause L)
        or IsSatisfiable(clauses with single literal clause not L) # Rule III
```

Figure 2 - Pseudocode for the DPLL algorithm

2.1.3.1 CDCL algorithm

The most significant development on top of DPLL is the conflict-driven clause learning (CDCL) algorithm, which is used by a significant number of competitive SAT solvers (Zaiken, 2017).

The CDCL algorithm is similar to DPLL in that it is a complete depth-first search, limiting the search space with unit propagation as per DPLL rule I, however where it differs is in the backtracking. In DPLL, the backtracking is strictly chronological and in single steps, as shown by the final return statement in the pseudocode (see Figure 2 above). If the first recursive function call (with single literal clause L) returns false, then the opposite assignment of “not L” is immediately tested.

In CDCL, the backtracking is non-chronological and can jump multiple steps, as rather than simply stating that there has been a conflict, it uses a dependency graph to calculate which variables were part of the conflict, and jump back to when they were assigned. This is done by “learning” additional clauses based on the conflicting variables, so that this path is not attempted again in the future. While this can have a huge performance improvement, it must also be weighed against the increasing number of clauses as the CDCL algorithm “can produce a huge amount of conflict clauses”, which must be managed and removed at regular intervals (Zaiken, 2017).

The process of reducing the number of these conflict clauses is where various CDCL algorithms diverge, including Zaiken’s particularly novel parallelisation approach called “NailSAT”. In this approach, rather than removing conflict clauses after a period of time, the conflict clauses are split into a number of overlapping sets. A new SAT instance is then spawned for each set of conflict clauses (with the original function clauses) and solved in parallel. The result of any of the spawned SAT instances is “sufficient to solve the original instance” (Zaiken, 2017). Given that the choice of conflict causes can greatly affect the computation time, this approach makes an effective SAT solver by increasing the probability of choosing the “right”

conflict clauses to follow, rather than having only a single instance which may have gone down the wrong search path near the beginning.

2.1.3.2 DPLL(T) algorithm

Another variation on the DPLL algorithm is the DPLL(T) algorithm, where the (T) represents an external theory which must be tested against in addition to the boolean clauses. This is not an unrestricted-SAT solver, but instead designed to solve SMT problems, which is an extension of SAT (see 2.1.2 Theme and variations).

The basic premise of the DPLL(T) algorithm is the same as DPLL, however an additional step is added before the final return statement of the pseudocode (see Figure 2) which calls a $Solver_T$. This $Solver_T$ may know more about the original objects (Atoms A) than the DPLL algorithm, which has only interpreted A as boolean values. The $Solver_T$ also knows about the Theory T and more importantly, how to test it. Upon testing, the $Solver_T$ has the ability to set remaining literals, add additional clauses, or backtrack as required (Ganzinger, 2004).

2.1.3.3 Branch and bound algorithm

In addition to the above variations, the DPLL algorithm can be combined with a branch and bound algorithm in order to minimise a cost function (Larrosa, 2009). This is used to solve the related weighted MAX-SAT problem as defined in Figure 1.

2.1.4 Local search algorithms

After DPLL-based algorithms, the majority of remaining SAT solvers employ local search algorithms. Local search algorithms are not complete searches, that is they are “not guaranteed to find a satisfying assignment… or prove unsatisfiability” (Moskewicz, 2001), however they can be faster to find a solution when one exists (Guo, 2013).

Local search algorithms work by iteratively improving the assignment of variables until all condition clauses are met, often involving random movements, such as in WalkSAT. In order to avoid being stuck in a local minima, clauses can be weighted with increasing weights when they continue to be unsolved.

Further to this approach, genetic or evolutionary algorithms can be used, such as GSAT. These are similar to local search algorithms in that they are incomplete, but they can be highly effective at reducing the search space and avoiding enumeration to find a solution (Pedrycz, 2002). Pedrycz et al’s Genetic-Fuzzy approach is one such example of this type of algorithm, whereby evolutionary optimisation is modelled with parameters for population size, number of generations, clones per generation and probability of mutation. This model produces values between 0 and 1, which are then bound to the nearest integer for the purposes of boolean output.

2.1.5 Portfolio algorithms

Although not truly a class of SAT solving algorithms in their own right, it would be naive to not mention portfolio algorithms, as they are among the top performers in recent years of SAT solving competitions, including the highly successful SATzilla (Nikolic, 2013).

Portfolio algorithms work on the principle that for a given SAT instance, solving time will vary significantly for different algorithms (Nikolic, 2013), thus having “availability of different solvers may be beneficial”. As such, a portfolio algorithm works by having available many solvers, and choosing the most appropriate solver for any given problem. How the SAT solver is chosen is the primary function of the portfolio algorithm itself.

For instance, SATzilla analyses the features of a SAT problem and predicts solving times based on “empirical hardness models obtained during the training phase” (Nikolic, 2013), which has been found to be a highly effective, but complicated process. Nikolic et al suggest that a better algorithm choice could be made if a “local, input-specific model” were used, based on a nearest-neighbour distance measure. Their implementation, called ArgoSmArT k-NN, was found to outperform SATzilla in all categories (Nikolic, 2013).

2.2 Artificial neural networks

Artificial neural networks (ANNs) are a class of algorithms for machine learning which model the inner workings of the human brain. In a biological neuron, synapses function as connection points which provide a number of input signals. If the net input signal passes a certain threshold, a pulse of electrical energy is created which propagates along the axon to the dendrite (Lorrentz, 2015). This action potential then reaches another synapse, providing an input signal to the next neuron, which may then fire based on the sum of its inputs. The human brain is made up of a large number of neurons which work together to enable complex thought.

An artificial neuron attempts to model the action of a biological neuron by having many inputs, each with a connection weight, and a single output. The output of an artificial neuron is calculated by taking a weighted sum of the inputs, adding a bias term, and evaluating an activation function of this value. An ANN is formed by the interconnection of many such artificial neurons, with the addition of raw data inputs and final outputs to the result which it is trying to calculate.

The final and most important aspect of an ANN is its ability to learn, that is to improve its result over time. When an ANN is first initialised, you can expect the output to be little better than random, but as the model is trained with training data, the accuracy of the results will improve until an acceptable quality is reached. There are various learning algorithms used, but the basic premise is that each piece of training data contains a set of input values and an expected output value. The input values are supplied to the model to calculate the actual output, which is then compared to the expected output. A numerical method, such as

stochastic gradient descent (SGD) is then used to calculate improved weightings on each connection in order to “make the loss function as small as possible” (Selsam, 2019).

While there are many well established machine learning tools available for data science applications, such as TensorFlow (Selsam, 2019), the following sections provide a high level overview of the basic principles and structures on which modern ANNs are based.

2.2.1 Multi-layer perceptron

A multi-layer perceptron (MLP) is the simplest form of neural network where neurons are separated into layers. The inputs to each neuron are strictly those neurons in the proceeding layer, such that no backtracking or interconnection within a layer exists. For this reason MLPs are also known as feed-forward networks (Selsam, 2019).

There are typically 3 layers in an MLP, an input layer which is directly connected to the input data values, the hidden layer, which takes its inputs from the outputs of the input layer, and the output layer which takes its input from the hidden layer and produces the output values from the model. This can be seen in the figure below.

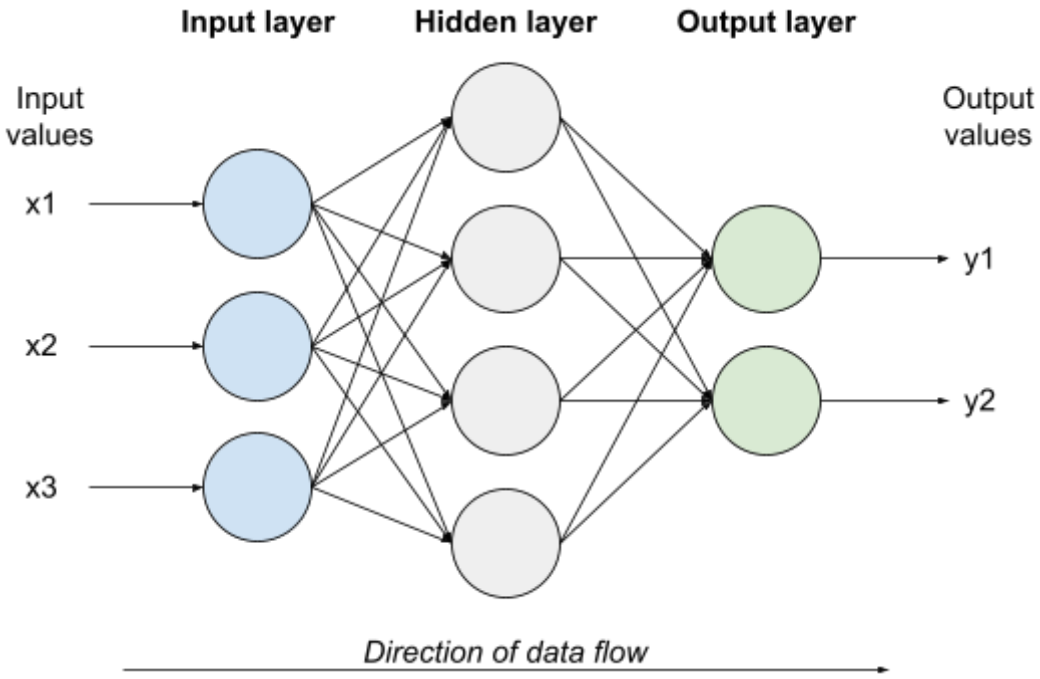


Figure 3 - Multi-layer perceptron structure

If more than 3 layers of neurons exist, they will be in the form of additional hidden layers between the input and output layers, but must follow the same rules of strictly only connecting to the layer immediately preceding them. Despite their simplistic model, MLPs have been shown to be good universal function approximators, even with only 1 hidden layer (Selsam, 2019).

The most common activation function for MLPs is rectified linear unit (ReLU) which can be represented algebraically by:

$$f(x) = x, \quad for\ x > 0$$
$$0, \quad for\ x \leq 0$$

Other common activation functions include the hyperbolic tangent $f(x) = tanh(a)$, which is normally used for the hidden layer, and the sigmoid, or logistic activation function $f(x) = \frac{1}{1+e^{-x}}$, often used for the output layer. There are also variants of the rectified linear unit, such as leaky ReLU, where $f(x) = 0.01x,\ for\ x \leq 0$, and exponential linear unit, where $f(x) = e^{x} - 1,\ for\ x \leq 0$.

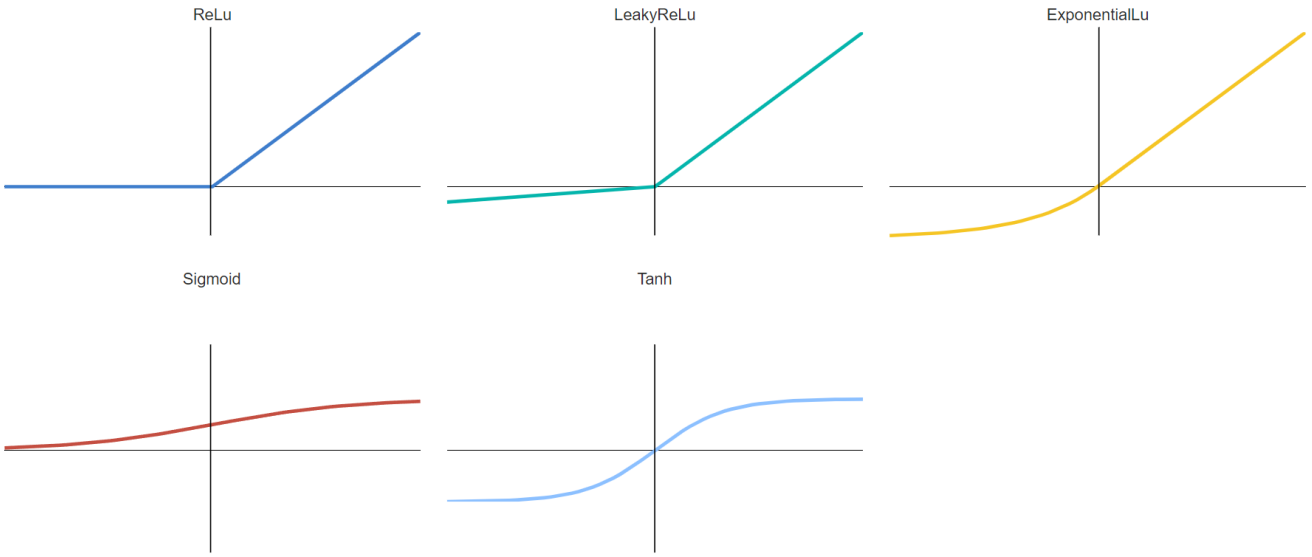


Figure 4 - Graphical representation of common activation functions

In order to train an MLP model, the weights of each connection between nodes are initialised to a small random value and the biases are set to a constant, such as +1 or -1 (Lorrentz, 2015). Back-propagation learning is then implemented by minimising the sum of the squared errors between the actual output and expected output, and calculating the derivative with respect to each weight.

One key limitation to MLPs is that the model's parameters, such as input vector length, must be defined upon initialisation before training the model (Selsam, 2019) restricting its usage to applications with only a fixed number of variables.

2.2.2 Recurrent neural networks

A recurrent neural network (RNN) differs from an MLP in that data is propagated forward and also backwards. In a recurrent network the input and hidden layers can accept feedback from other layers in the network or external environmental data (Lorrentz, 2015). In fact, the structure does not require the segregation of neurons between input and hidden layers at all (Alway, 2021).

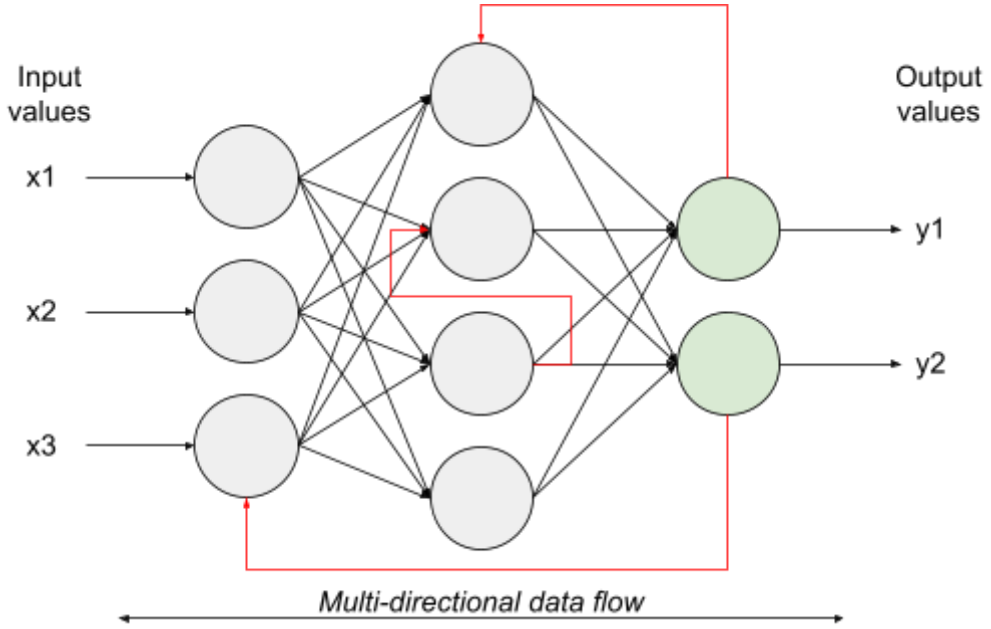


Figure 5 - Recurrent neural network structure

A feature of RNNs is the ability to process a sequence of inputs of dynamic length, unlike MLPs where input vector length must be predefined, making them useful for a wide range of applications. This type of model is said to have state, stored in the hidden layer, allowing it to “remember” previously seen values and exhibit dynamic behaviour with respect to a time domain (Selsam, 2019).

One such example of a recurrent neural network is the Hopfield network, in which neurons output only the binary values 0 and 1, based on a threshold function of the inputs. At any time the state of the machine can be expressed as a binary word of length N, where N is the number of neurons in the network. The output of each binary neuron is fed back as an input to all neurons, including itself, meeting the definition of a recurrent network (Alway, 2021).

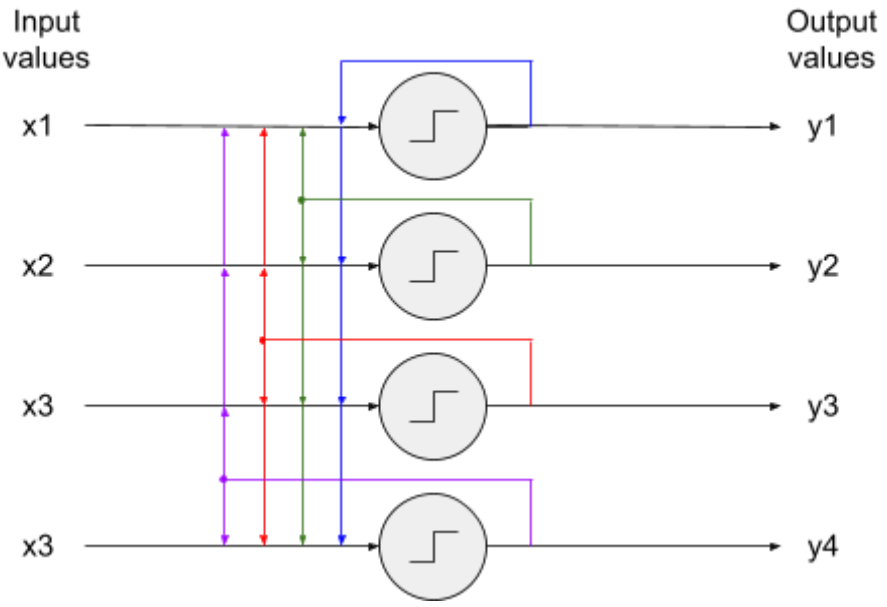


Figure 6 - Hopfield network structure

2.2.3 Graph neural networks

Although unlikely to be useful for this particular application, it is worth discussing for completeness a third key type of ANN, the graph neural network (GNN). In recent years, GNN based algorithms have found many practical uses in modelling large social connection graphs and is considered to potentially “bridge the gap between machine learning and symbolic reasoning” (Liu, 2020).

Complex graphs, with a lot of interconnection, are poorly represented by RNN models as they introduce long sequences when unrolled, posing major challenges to SGD-based learning algorithms (Selsam, 2019). It is, however, easy to understand and design a neural network which can represent large graphs using a GNN.

Unlike an MLP, which takes a fixed number of input variables, or RNN, which takes a sequence of variables, a GNN represents each node of a graph by a fixed size vector, called the embedding of a node. This vector is then used in a simpler ANN, such as MLP, which specifies how a node updates its embedded value based on an aggregation of the embedded values of its neighbours in the graph (Selsam, 2019).

One particularly interesting application of the GNN structure is in Ozolins’ “goal-aware” algorithm for solving the boolean satisfiability problem discussed at length in section 2.1 Boolean satisfiability problem. Their proposed QuerySAT algorithm uses the internal structure of GNN to calculate a loss function for each clause based on a query point. This approach is found to outperform GSAT, a comparable incomplete local search algorithm on the SHA-1 preimage attack task (Ozolins, 2021).

2.2.4 Learning to rank (LTR)

Learning to rank (LTR), also known as machine-learned ranking (MLR), is an application of machine learning to evaluate the relevance of an item in information retrieval problems. This is not to say that LTR is strictly part of the study of ANNs, but it is included here due to the high level of success which neural network based algorithms, such as RankNet, achieve in solving such problems (Wang, 2017).

A typical example of a problem in LTR is, provided a query, to rank search results based on their importance to that query. As such, many problems in information retrieval can be solved by LTR, whether it be search results or other “commercial applications such as news feed content ranking, recommendation engines (or) machine translation” (Goswami 2018).

There are 3 main approaches to ranking algorithms; pointwise, pairwise and listwise. The pointwise approach judges each individual item on its merits and determines a specific value of relevance against the query, without considering its comparison to others. ANNs used for regression and classification are good examples of algorithms used for the pointwise approach (Goswami, 2018).

As the name implies, the pairwise approach compares 2 possible results, and determines which of them is more relevant to a given query. This is the approach taken by the RankNet algorithm, which uses an ANN to calculate the probability that one search result is better than another by inputting a vector of features (Wang, 2017).

The listwise approach is the most general, and attempts to sort the entire list of search results in order of importance as one process. This approach is applied in Wang's attention based algorithm, using a RNN structure with weights that can change over time. This differentiates their algorithm from RankNet, and other similar algorithms, by learning the "matching mechanism between a query and search results with a similarity matrix" (Wang, 2017).

2.3 Critical review of previous software

The process of casting is an interesting mixture of numerical analysis of ability marks and a human factor of what constitutes the "best" applicant for a role. While there is a strong correlation between ability marks and final role allocation, it does not tell the whole picture, something I have seen first hand with many years of casting for Cumberland Gang Show. The previous software package, called the Casting Expert, was written in house and attempts to suggest applicants based on a heuristic algorithm. The history and shortfalls of this approach are described below.

2.3.1 Casting Expert for DOS (<= v4.1)

Between 1985 and 2006, a set of two DOS programs, written in Pascal, were used for the cast selection and role casting processes respectively. First used in 1985, v2.0 allowed electronic selection of 144 cast members from as many as 200 applicants, and the casting of over 200 character roles. This was a huge step forward in the mostly administrative process of cast selection, in which the weighted average "overall" mark was calculated digitally, the required number of cast members in each cast group could be selected, or reselected quickly, and reports could be generated following any changes made. Previously all of those steps were done on paper, requiring a great deal of manual labour to type and copy the result, often with human error introduced and rework required.

Figure 7 - An example of the whiteboard previously used for casting

In terms of role casting, the above was doubly true. Previously done on a large whiteboard grid, it was complex to understand, hard to visualise, and easy to make mistakes. However the improvement at this stage was mostly administrative, as the process (and even UI) of the new DOS application closely follows that of the casting board, as can be seen in the “Add/Alter Casting” screen, which looks exactly like a single row of the old casting board grid. Despite its lack of “intelligence”, this system did however provide one significant advantage- the ability to run verification checks. Specifically, it checks that every character has the required cast, every person is in exactly 1 item per bracket, and that no person is in consecutive items.

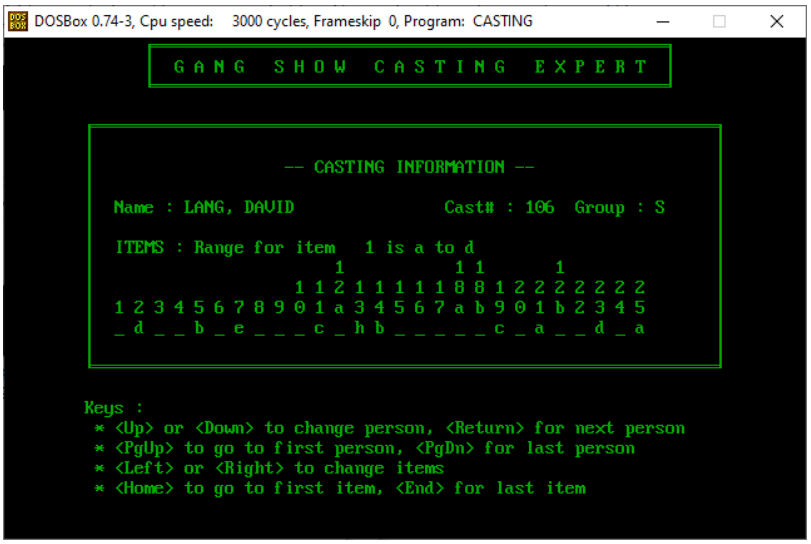


Figure 8 - A screenshot from Casting Expert v4.1

It wasn’t until 1990 that version 3.3 of the role casting application included intelligent recommendations by an expert system, and as such the name “Casting Expert” was born. The expert system in question operated as a heuristic expert system, embedding specific business logic about the structure of the show and many years of hard-earned domain knowledge from the production team. This saw the Casting Expert calculate a suitability score for each potential cast member for a specific role, and rule out those which were otherwise inapplicable (for example if they were already cast in this bracket, or were in the previous item). The application had minor tweaks each year but the expert system remained largely unchanged in its structure until the final Pascal version, version 4.1, was written in 2003. It was used until 2006, when it was replaced by the Windows version.

A major limitation of this version of the application was in the data layer. The data was stored in a series of text and binary files, of which there were inputs, internal process links, and outputs. A program crash could cause corruption, which in turn made all files unreadable, so regular backups were kept. Also, any changes to the inputs invalidated much of the casting data, so this was very much a one-way process. If a mistake was found during the casting process, files would have to be edited manually, or sometimes simply started again. Similarly if any fields were changed or added between years, it would stop previous year’s data from being opened.

2.3.2 Casting Expert for Windows (v4.2-v5.3)

In 2006, the casting expert system was re-written for Windows in Visual Basic 6. This was a significant step forward, solving the major limitation of data structures with the use of a well-formed relational database, and bringing the casting expert out of the dark ages of console applications into the bright new world of graphical user interfaces (GUIs). This was fortuitous timing, as it turned out that Windows XP was the last version of Windows which could run the previous DOS applications natively. They were incompatible with the successor, Windows Vista, which was released in the same year, and have since only been able to run through emulation.

With this new GUI and properly structured data storage, a number of new features could be added, including storing and displaying photos of each cast member, importing/exporting of applicant's personal details, and the ability to make changes to items, character roles and their requirements during the casting process. In addition to this, the data was stored in a single Microsoft Access MDB file, which could be easily backed up or copied between computers. The transactional nature of SQL also meant that data was saved after each action and no data corruption could occur, even in the event of a program crash.

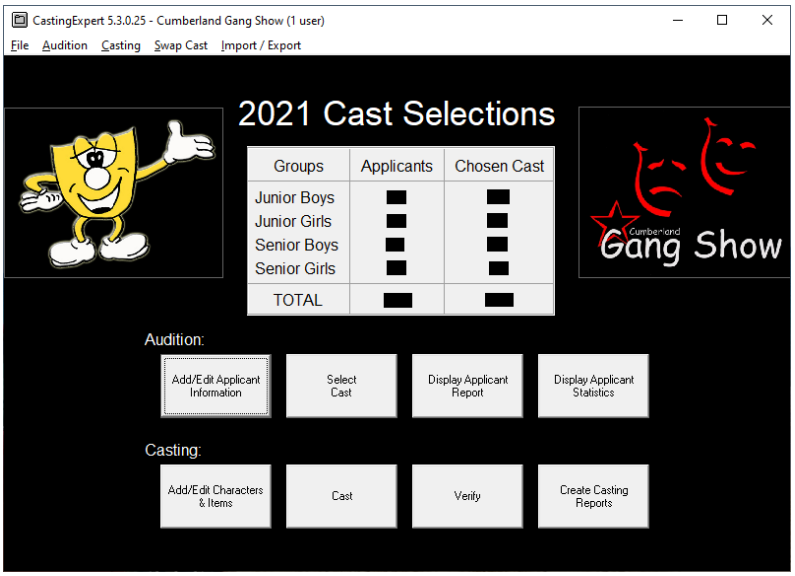


Figure 9 - A screenshot from Casting Expert v5.3

However, even with a complete rewrite from scratch, and all those improvements aside, the casting process and expert system remained largely unchanged. The intelligent recommendations still operated with the same heuristic algorithm of the previous DOS versions, calculating a suitability score, and suggesting the highest applicable cast member. Balancing talent between cast groups, or groups of characters, had to be done manually- a tedious and argumentative process between production team members with differing goals. There were also external factors which needed to be considered, such as keeping siblings in the same cast groups, which were not automated. Further to this, the system did not learn, improve or adapt, and if we decided not to choose a person for a role, the expert system would continue to suggest them for every role after this, until we did, simply because it calculated that they were the most suitable applicant.

Anecdotally, users often felt like they were fighting the expert system, which had a favourite candidate that we simply did not want to choose.

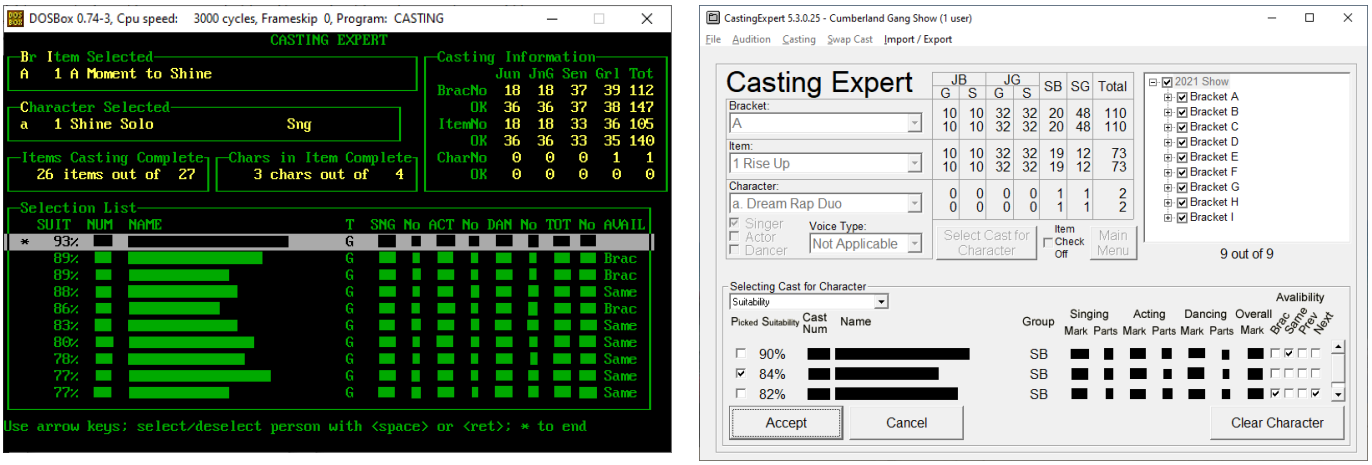


Figure 10 - A comparison of the DOS and Windows user interfaces

The lack of change in process is also highly evident in the UI. Despite being a Windows GUI application rather than a DOS console application, the new version bears a striking resemblance in UI design, as seen in Figure 10 above. This is not by coincidence, but simply because it was designed to be the same system, just in a new language, for a new platform, with some additional features. In addition to the lack of innovation in UI, the major limitation of this version is now in its intelligence. What was a ground-breaking heuristic expert system in 1990, can only be described as antiquated and simplistic in 2021, some 30 years later.

3. Methodology

In order to answer the question of whether SAT solving and machine learning algorithms can improve the quality of an automated casting system, I will use the following quantitative methodology.

1. Create a persistent data model to represent the domain of theatrical performances
2. Implement a functional GUI application, abstracting the recommendations and other major intelligent features to external “casting engine” interfaces
3. Implement heuristic engines, utilising the same basic logic as the previous CE applications
4. Implement SAT solving engines, utilising a SAT solving algorithm
5. Implement neural network engines, utilising an ANN model
6. Automate the cast selection process to emulate a user accepting the recommended selection, using all years of the CGS historical casting data (from 1993 to 2021)
7. Analyse each output selection to determine the balance of talent between alternative casts
8. Compare the balance for each SAT engine to the heuristic engine and human produced results
9. Train ANN models using even years of CGS historical casting data (from 1994 to 2020)
10. Automate the casting process to emulate a user accepting the recommended cast for each role, for the odd years of CGS historical casting data (from 1993 to 2021)
11. Analyse each output casting to determine the accuracy of recommendations compared to the actual casting produced by the CGS production team
12. Compare the accuracy of each ANN engine to the heuristic engine baseline

A detailed description of the work completed in steps 1-8 and 9-10 can be found in the sub-sections below. The remaining steps produce the results which are analysed in section 4. Analysis.

Following the quantitative analysis above, an MVP of the casting software will be produced using the best combination of “casting engine” implementations, and evaluated by a subset of the CGS production team to gain their feedback. This will be compared to the project goals (see section 3.2 Project deliverable), the functional requirements (see Appendix B - Functional requirements), and the list of previously requested features and improvements (see Appendix A - Previous user feedback).

This qualitative analysis is expected to provide a representative sample of what the production team will experience between December 2021 and January 2022 as they prepare for the July 2022 show using the new software package.

3.1 Hypotheses

While the heuristic algorithm calculates weighted average marks perfectly, what it fundamentally lacks is the human factor. I believe that SAT solving and machine learning will allow the casting engine to bridge the

gap from purely numerical analysis to a more creative output, learning the human factor over time from interactions with the user.

Formally put my hypotheses are as follows;

1. Using a SAT solving algorithm will produce a more even balance between alternating casts, than a heuristic algorithm
2. Using a neural network will produce a more accurate recommendation of applicants for a role, than a heuristic algorithm
3. Using a neural network will result in a more even spread of parts between applicants of a reasonable ability level, than a heuristic algorithm

3.2 Project deliverable

A key deliverable of this project will be the next software package for the cast selection and role casting of Cumberland Gang Show, but it will not be the next version of the Casting Expert. It will be generic for the casting of any theatrical production, rather than specific to Gang Show. It will have a modern user interface, and lead the user through a priority-first casting process, rather than chronologically through the items. It will allow balancing talent between groups and encourage a fair distribution of parts between talented applicants. But none of those are the overall goal of this project.

The goal of this project will be to improve the intelligent recommendations of the casting system. Given that the casting process is inherently a mix of hard (boolean) rules, and soft (weighting) rules, I hypothesise that using a combination of SAT solvers and neural networks will provide an increase in accuracy of casting recommendations.

The new software package will be called “CARMEN: Casting And Role Management Equality Network”.

3.3 Development environment

Due to the user requirement of a desktop GUI application, the following development environment and language features were chosen.

- Visual Studio 2019 (Community Edition) IDE
- Primary programming language C#9
- Targeting .NET 5 SDK
- Entity Framework Core 5.0.8 ORM
- Windows Presentation Foundation (WPF) GUI framework

The full source code of the completed application is available at https://github.com/davidlang42/CARMEN, under the GPLv3 license.

3.4 Data model

The following data model has been designed and implemented using EF Core, an Object-Relational Mapper, to map POCO classes to a relational database structure. The model can then be stored as a single file SQLite3 database with ".db" file extension, or hosted on a server using any major SQL database provider, such as PostgreSQL or MariaDB.

In order to represent the domain of a theatrical production, the "show model" includes the following primary objects.

- Show structure nodes (Show, Section, Item, Role)
- Applicant details (Applicant, Marking Criteria)
- Cast structure (Cast Groups, Alternating Casts, Tags)
- Requirements (based on Marking Criteria, Applicant details)

See Appendix C - ShowModel data descriptions for implementation details.

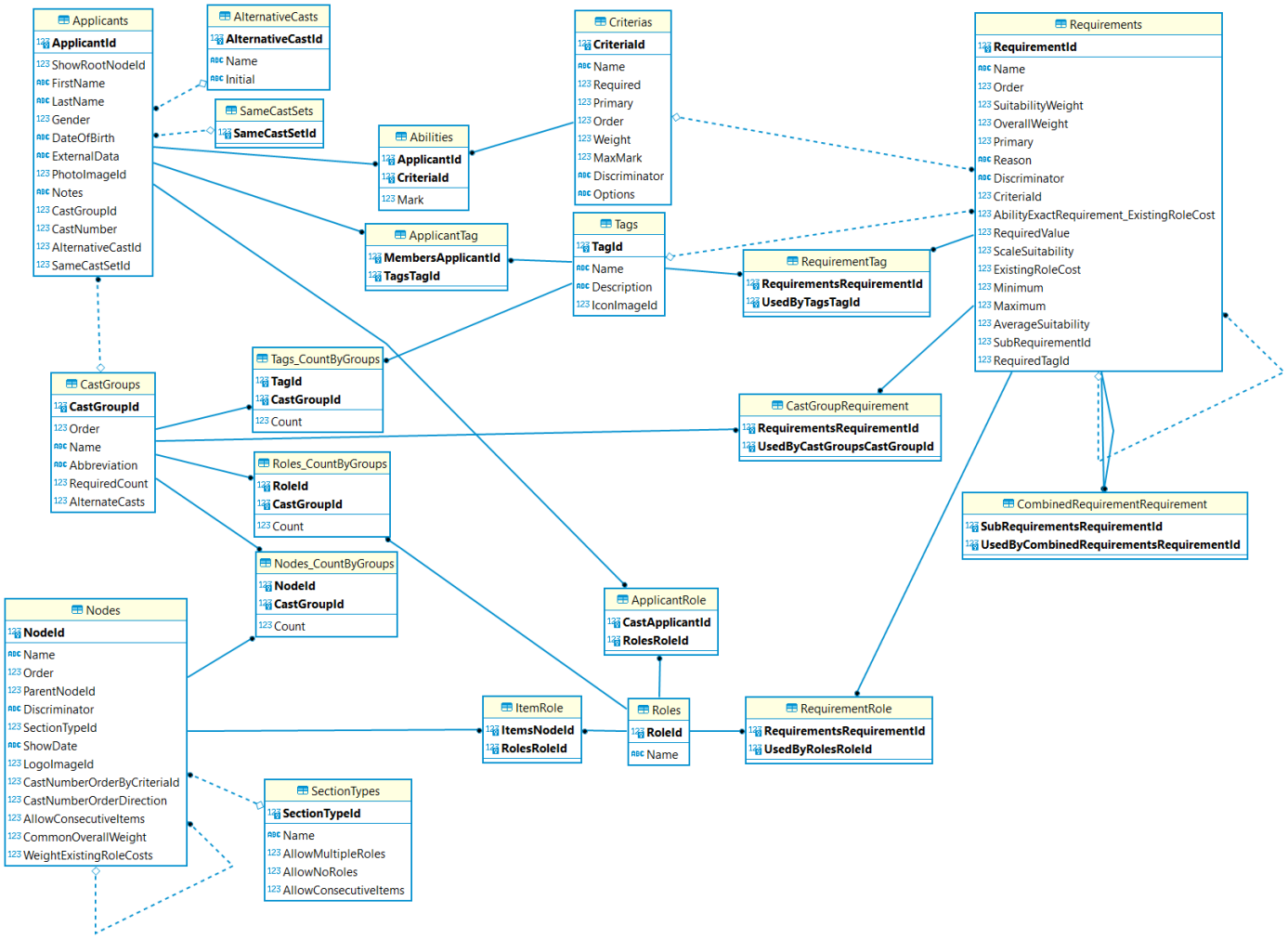
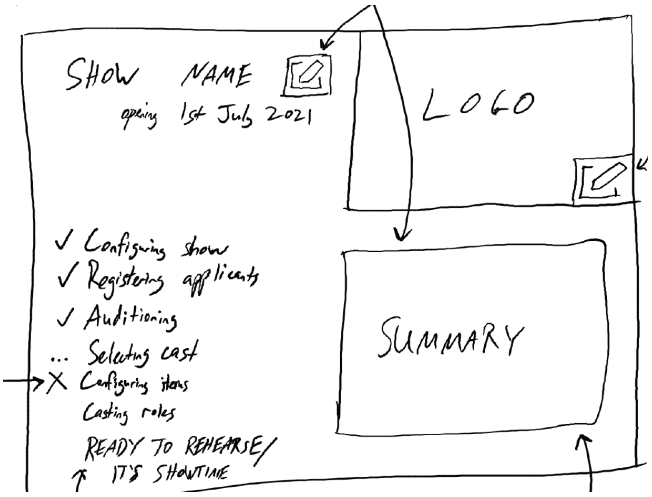


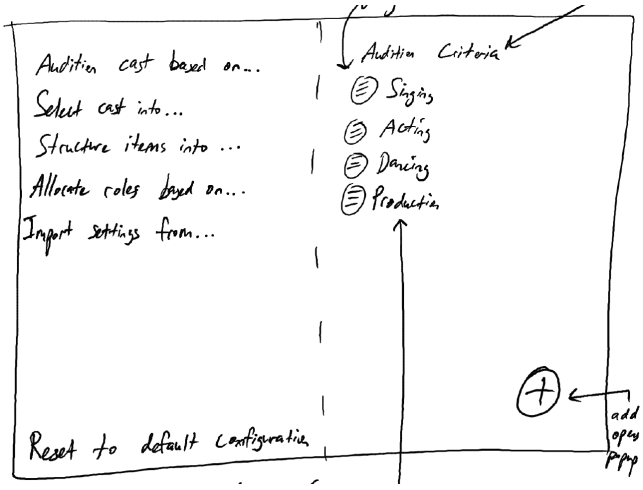
Figure 11 - Relational structure for the Show model

3.5 Functional GUI implementation

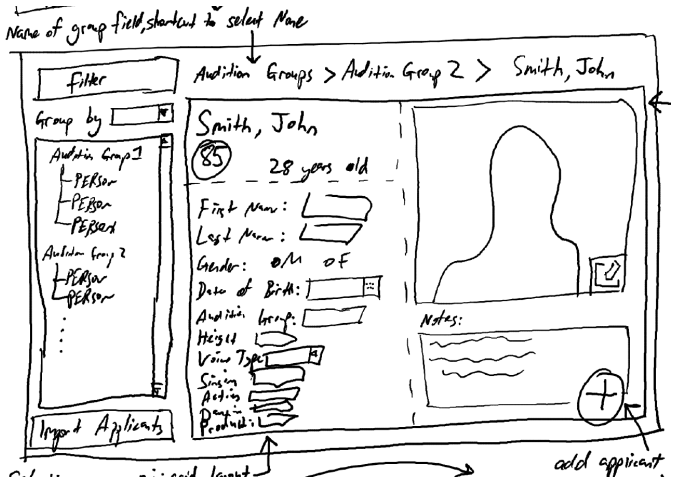
I have implemented a functional UI that is feature complete for the purposes of configuring and casting a show. This includes basic CRUD functions for each part of the data model, as well as data persistence, verification checking, and the end to end user process. At this stage the GUI is not perfect, fancy, or even foolproof, but it does allow access to all functions of the CARMEN software package. A draft specification can be seen in the storyboards in Figure 12 below.



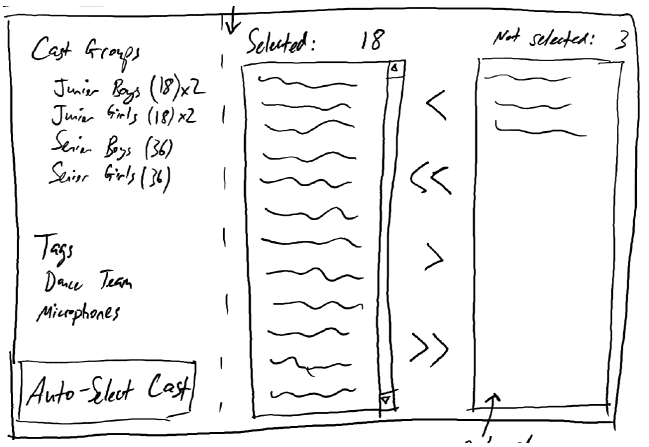
Main menu / status checks



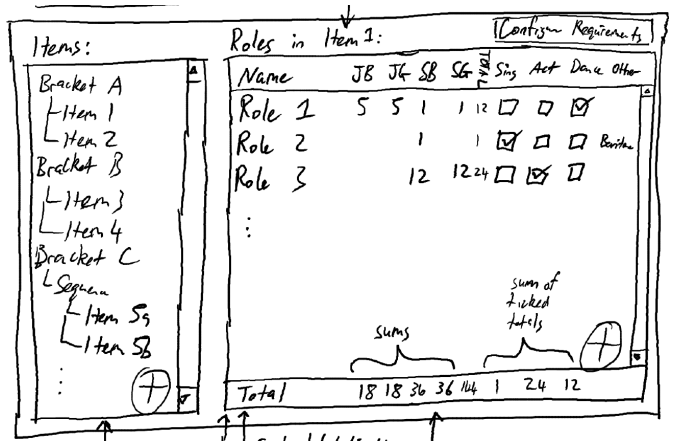
Manage show configuration settings



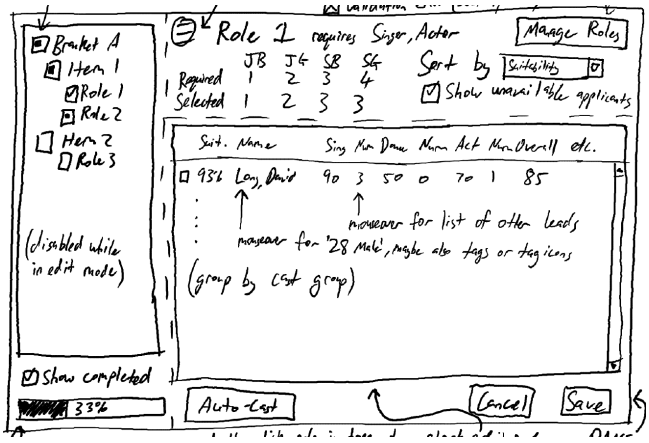
Edit applicant details



Cast selection



Create show structure/items/roles



Allocate roles to applicants

Figure 12 - Storyboards for CARMEN user interface

3.6 Abstraction of the casting engines

Although the functional creation of a fully functional application was a time consuming process, it was vital to complete this in order to identify the points in the process where the underlying casting engines will be engaged.

This allowed the casting engines to be implemented as abstracted interfaces separate from both the UI and data model, and only dependent on the data model itself. By doing this, the various implementations of each type of casting engine are easily swappable for analysis and comparison.

The full interfaces are provided in Appendix D - Casting engine interfaces, but it is sufficient to understand that there are 3 types of casting engine;

- An “audition” engine, which performs operations relating to the auditioning of applicants, such as the calculation of an applicant’s overall ability level,
- A “selection” engine, which performs operations relating to selecting applicants into the cast, such as selecting cast groups, balancing alternative casts, applying tags, and allocating cast numbers,
- An “allocation” engine, which performs operations relating to the allocation of roles to cast members, such as recommending cast for a role, automatic balancing of cast between roles, and determining the ideal order to cast roles.

3.7 Implementation of the heuristic algorithms

In order to gain a baseline in section 4. Analysis, heuristic engines have been implemented to meet the engine interfaces with similar functionality to the existing CE software.

The relevant implementations can be found in Appendix E - Heuristic engine implementations, but an overview of the heuristic engines, including any differences from the original heuristic algorithms of CE, is included below.

3.7.1 Audition engine

The heuristic algorithm implemented for the audition engine is called WeightedSumEngine. This engine calculates the overall ability of an applicant as a weighted sum of their individual criteria marks. This is the same calculation as was done by CE, except the weighting can be set by the user rather than being hard coded at predetermined values. Although not used in my testing, this implementation improves upon CE by allowing negative weights if desired.

3.7.2 Selection engine

The HeuristicSelectionEngine, and in fact all selection engines, select applicants into cast groups in order of their overall ability (as calculated by the audition engine). Each applicant is checked against the

requirements for the cast groups, and is placed in the first eligible cast group which has not yet reached the required number of cast members. Although not used in my testing, this implementation improves upon CE by respecting previously selected cast members, where the old algorithm would clear all applicants' cast groups before re-selecting. Another important difference is that CE would skip selecting the last applicant in a cast group if there were not an even number of eligible applicants, whereas this implementation will select all available.

Similarly, the process of applying tags is common to all selection engines, and is done in order of the applicant's suitability for that tag. If applicants have the same suitability, the applicant with the higher overall ability is chosen first, which is an improvement on CE. Tag suitability is calculated as the average of the suitability for each requirement of the tag. Tags are applied to applicants until the required number of each cast group has been tagged, or until no more applicants satisfy the requirements of the tag (which is important for tags with no required count). Although not used in my testing, this improves upon CE by allowing multiple custom tags (previously a single hard coded tag only), allowing tags to have multiple requirements, and allowing tags to require other tags to be applied first (provided that there are no circular dependencies).

The selection engine also allocates cast numbers to applicants which have been selected into the cast. The implementation is common to all selection engines, and numbers the selected applicants in the order of their marks for a chosen criteria, in a chosen direction. This improves upon CE by being customisable, rather than a hard coded criteria and direction, however unlike CE it does not allocate negative numbers to rejected applicants and instead just leaves them blank.

The HeuristicSelectionEngine differs from other selection engines in how it balances selected cast members between alternative casts. The concept of alternative casts is that there are multiple sets of people who perform the show at different performances. Each alternative cast will have a complete set of cast members allocated to the roles, and alternative casts are mutually exclusive (for cast groups which have “alternate casts” enabled). Therefore it is important to balance talent between these casts to enable a reasonable allocation of roles later. The heuristic approach here is rather basic, firstly assigning the alternative casts of those in same cast sets (that is, cast members which should be kept in the same alternative cast) and then ordering by cast number and allocating to the alternative cast with the lowest number of members at the time. This effectively mimics what CE would do, except that CE did not have a concept of same cast sets, and did not respect previously set alternative casts.

3.7.3 Allocation engine

The heuristic algorithm implemented for the allocation engine is called WeightedAverageEngine, as it calculates the suitability of an applicant for a role as a weighted average of the applicant’s overall ability and their suitability for each requirement of the role. Before averaging, the suitability for each requirement is reduced by a number of marks based on the number of lead roles already allocated to the applicant. This is

identical to the previous CE implementation, except that the weights and existing role reduction factors are not hard coded and can be changed by the user. There are also additional options previously not available including:

- The option to subtract the existing roles costs AFTER averaging rather than before,
- Count existing roles as partial roles if more than 1 requirement is selected,
- Weight the overall ability by a different factor per requirement, rather than a single value.

The allocation engine then recommends applicants for roles based on this suitability calculation, in a procedure which is common to all allocation engines. As well as respecting previously cast applicants (which CE did not do), this procedure checks the availability of eligibility of a cast member for the role. This is a significantly more advanced version of what CE would check because roles can now be in multiple items rather than just one, and applicants can be ineligible for a role even if their suitability is non-zero. The procedure also attempts to keep applicants with the same cast number allocated to the same roles, when a role has no specific requirements.

The allocation engine can also be used to cast multiple roles at once, balancing talent between them. This operation is common to all allocation engines and works by listing all eligible applicants for each role in order of suitability. Each role then gets to take their next available applicant, one at a time, and is repeated until all roles are cast or no applicants remain. There is a special case that if only just enough eligible applicants remain for a role, it will take them all rather than just one. This is helpful to handle cases where one role has significantly less eligible applicants than the others.

The allocation engine is also responsible for determining the order roles should be cast, which is also common to all allocation engines. Roles are grouped first by structural segments within the show, then into tiers based on the priority of their requirements. Within each tier, roles are ordered by the least required cast first, then by the smallest number of eligible cast available. By default the structural segments are the sections within the show where a cast member can not have multiple roles, but there is an option to instead consider the whole show as one segment. There are 3 options for determining requirement priority tiers;

1. All requirements at once- resulting in 2 tiers
    a. Roles with any requirements
    b. Roles without requirements
2. Primary requirements first- resulting in 3 tiers
    a. Roles with primary requirements
    b. Roles with only non-primary requirements
    c. Roles without requirements
3. Individual requirements in order- resulting in N+1 tiers (N is the number of unique requirements)
    a. Roles requiring only the 1st requirement
    b. Roles requiring the 2nd requirement (and maybe the 1st requirement)
    c. ...

d. Roles requiring the Nth requirement (and maybe any earlier requirements)
e. Roles without any requirements

3.8 Implementation of the SAT solving algorithm

In order to test hypothesis #1, that SAT solving algorithms will produce a more even balance of talent between alternative casts, a number of selection engines have been implemented which make use of SAT solving algorithms. The approach taken by each engine for balancing casts is described below. All other selection engine operations use the common implementations described in section 3.7.2 Selection engine.

These six alternatives are evaluated and compared in section 4. Analysis to determine which approach is best at balancing talent between alternative casts in the cast selection process.

3.8.1 Chunked pairs

The ChunkedPairsSatEngine approach balances talent between casts by sorting the applicants in descending order of their marks for a certain criteria, then pairing applicants down the list (ie. the highest mark is paired with the 2nd highest, 3rd highest is paired with 4th, etc). This is done for each of the primary criterias. A boolean expression is generated with clauses requiring that the applicants of each pair are in different casts. Additionally, clauses are added to require applicants in a same cast set to all be in the same cast. This requires a special case while pairing, that applicants in the same same cast set are never paired together. In this case the 2nd applicant which would have been paired is skipped, and used in the next pair. Finally, unit clauses are added to specify existing alternative cast assignments.

This completed expression is then solved by a DpllSolver, which is my implementation of the DPLL algorithm (see section 2.1.3 DPLL-based algorithms) to solve k-SAT problems using unit clause propagation, pure literal propagation, and branching by the first remaining literal of the first remaining clause. There is no doubt that my implementation of the DpllSolver is inefficient and could be improved with CDCL, however it was not required as this approach usually completes in less than a second.

If the expression is found to be unsolvable, which it usually is with multiple primary criteria, the approach is repeated but with a chunk of size 4 rather than pairs (of size 2). In these chunks, it is required that half the applicants are in one cast, and the other half are in the other cast. This is repeated, increasing the chunk size by 2 each time, until a solution is found.

3.8.2 Top pairs

The TopPairsSatEngine approach is similar to ChunkedPairSatEngine, except if the expression is unsolvable, the chunk size stays at 2 (pairs) and the number of pairs is reduced by 1. By omitting the least talented pairs, this will prioritise even balancing of cast where it matters most, at the top of each criteria.

The completed expression is solved by a DpllSolver, however there is technically no guarantee that the resulting alternative cast assignments place the same number of applicants in each cast (which is a domain requirement).

3.8.3 Hybrid pairs

The HybridPairSatEngine approach is a combination of chunked pairs and top pairs. Initially it runs like the top pairs approach, pairing applicants and reducing the number of pairs until it is solvable. Once it has a solvable set of pairs, the hybrid pairs approach keeps those pairs and tries to chunk the remaining applicants not paired with a chunk size of 4. Once again the number of chunks is iteratively reduced by 1 until this is solvable, then those chunks are kept and the remaining applicants not chunked are chunked with a chunk size of 6. This is repeated until the chunk size is larger than the remaining applicants not chunked, then the most recent solution is kept.

This has the effect of getting the same outcome as TopPairsSatEngine for the top applicants, but then still having some balancing at the lower end of the spectrum. It also alleviates the issue of splitting the cast unevenly between alternative casts which could have occurred in the top pairs approach. This approach also solves the completed boolean expressions using a DpllSolver.

3.8.4 Three's a crowd

The ThreesACrowdSatEngine approach lists the applicants in the same way the pairs based sat engines, but takes overlapping sets of 3 applicants rather than taking mutually exclusive pairs (ie. the first set is {1st, 2nd, 3rd}, then {2nd, 3rd, 4th}, then {3rd, 4th, 5th}, etc). Boolean clauses are then generated to require that each set of 3 is not all in the same cast (ie. there is at least 1 which is different). Similarly to the pairs based approaches, these clauses are combined with the same cast set clauses and existing assignments to become the final boolean expression.

The advantage of this approach is that in being a much less strict set of rules, a fallback is not required. However the disadvantage is that it will not necessarily end up with the same number of cast members in each alternative cast, which is a domain requirement.

Due to this external domain requirement, this becomes an SMT problem rather than unrestricted-SAT and is therefore solved with a DpllTheorySolver, which is my implementation of the DPLL(T) algorithm (see 2.1.3.2 DPLL(T) algorithm) to solve SMT problems using unit clause propagation, branching and theory testing. The exclusion of pure literal propagation is necessary for ALL-SAT solvers as it may skip a valid solution in trying to find any solution the fastest. In the case of DPLL(T), it is possible for that skipped solution to be the only one which meets the required theory.

3.8.5 Rank difference

The RankDifferenceSatEngine approach attempts to balance the alternative casts by minimising a cost function, that is, the difference in the sum of ranks. In this approach, the applicants are listed in increasing order for a certain criteria and assigned a rank starting at 1 and increasing as the applicant's mark increases, with duplicate marks receiving the same rank. This is done for all primary criteria.

Now that the ranks are known for each applicant, the rank difference can be calculated by summing the ranks per criteria in each cast, and finding the difference. The absolute difference for each criteria is then summed to get the total rank difference. In order to ensure that the resulting alternative casts have the same number of cast members, the rank difference cost function is special cased to return maximum cost if the cast member counts are not equal.

The SAT clauses in this approach come only from the same cast sets and existing assignments, which when combined with the rank difference cost function, becomes a weighted MAX-SAT problem. This is solved by a BranchAndBoundSolver, which is my implementation of the branch and bound algorithm (see section 2.1.3.3 Branch and bound algorithm) combined with an ALL-SAT solver. As SAT solutions are found, the upper and lower bounds of the rank difference cost function are calculated, until they converge on the optimal solution.

Due to the sheer number of solutions available, calculating the cost function for the full solution space is impractical. The BranchAndBoundSolver is therefore configured to terminate upon finding a solution with an average of 1 or less rank difference per criteria, or when more than 5 seconds have passed without an improvement.

3.8.6 Best pairs

The BestPairsSatEngine approach is similar to the top pairs approach in that it pairs down the lists and reduces the number of pairs until it is solvable. However it uses a BranchAndBoundSolver with the rank difference cost function, rather than the standard DpllSolver, to find the solution with the lowest rank difference from the solutions available to the top pairs approach.

This has potential advantages over both the top pairs and rank difference approaches, whereby it maintains the priority of the top applicants from top pairs, whilst reducing the solution space of rank difference.

3.9 Implementation of the ANN model

In order to test hypothesis #2, that a neural network algorithm will produce more accurate recommendations for a role, a number of allocation engines have been implemented which make use of neural networks. The approach taken by each engine for finding recommended applicants for a role is

described below. All other allocation engine operations use the common implementations described in section 3.7.3 Allocation engine.

The three allocation engine approaches are evaluated and compared in section 4. Analysis to determine which provides the best recommendations when allocating roles to cast.

Additionally, an audition engine which makes use of a neural network has been implemented. Although it is not explicitly evaluated, the approach taken for determining an applicant's overall ability is described below for completeness.

3.9.1 Neural audition model

The NeuralAuditionEngine models the overall ability of an applicant as a single-layer perceptron (SLP), effectively performing the same weighted sum calculation as the WeightedSumEngine described in section 3.7.1 Audition engine. The difference with the neural approach is that when the user manually selects cast, it trains the neural model to determine new weights for each criteria and prompts the user to update them.

This engine uses the stochastic gradient descent (SGD) training algorithm and allows configuration of the following parameters;

- Loss function- the function used to calculate the loss of the output, compared to an expected output, which is then propagated back to change the neuron's input weights.
- Learning rate- the factor which determines how much the weights change based on the loss calculated in one iteration (aka the speed at which it learns).
- Training iterations- the maximum number of training iterations which are performed per user operation.

The inputs to the neural network are the marks for each criteria of two applicants, with the output determining which of the applicants is better. A discussion of why the pairwise approach was used rather than the pointwise approach can be found in section 4.2.1 Pointwise vs pairwise.

Given that this is a classification task, the sigmoid activation function is used to bound the output between 0 and 1, where a value greater than 0.5 indicates that the first applicant is better.

3.9.2 Role learning allocation model

The RoleLearningAllocationEngine models the suitability of an applicant as a SLP, effectively performing the same weighted average calculation as the WeightedAverageEngine described in section 3.7.3 Allocation engine. The difference with the role learning approach is that when the user manually allocates a role to cast members, it trains the neural model to determine new weights for requirements relevant to the role and prompts the user to update them.

Similarly to the neural audition model, the role learning allocation engine uses the SGD training algorithm with the same configuration parameters described in section 3.9.1 Neural audition model. There is one additional engine parameter for the RoleLearningAllocationEngine;

- Reload weights- an option for when to reload the requirement weights into the neural network, of
    - Always- after every training operation, reload the neural weights to remove small changes which were not significant enough to change the requirement weights
    - On change- when a neural weight change is significant enough to change the requirement weights, reload the neural weights to remove small variations in applicant A and B weights (which would normally be of the same magnitude but opposite polarity of each other)
    - Only when refused- let the neural network keep its minor variations in case they eventually become significant after repeated training operations, so only reload the neural weights when a change has been actively rejected by the user

The inputs to the neural network are the individual requirement suitabilities for two applicants, as well as their overall abilities, with the output determining which of the applicants is more suitable for the given role. A discussion of why the pairwise approach was used rather than the pointwise approach can be found in section 4.2.1 Pointwise vs pairwise.

Given that this is a classification task, the sigmoid activation function is used to bound the output between 0 and 1, where a value greater than 0.5 indicates that the first applicant is more suitable. The options and default values for the other parameters are discussed in section 4.2.4 Default engine parameters.

3.9.3 Session learning allocation model

The SessionLearningAllocationEngine works in much the same way as the role learning approach (see section 3.9.2 Role learning allocation model), but by training the neural network with multiple roles at once, rather than just one.

To achieve this, an additional engine parameter is used;

- Training schedule- an option for when to perform training operations, of
    - Once per session- at the end of a casting session, train the neural network once with training data from all of the roles which were allocated cast during the session
    - Every role individually- every time cast are allocated to a role, train the network with training data from that role only (this differs slightly to the role learning approach in that all requirement weights may be changed, not just those relevant to the current role)
    - Every role stockpiling- every time cast are allocated to a role, train the network with training data from all roles which have been allocated cast during the session up to that point

The options and default values for the various parameters are discussed in section 4.2.4 Default engine parameters.

3.9.4 Complex network allocation model

The ComplexNeuralAllocationEngine models the suitability of an applicant as a feed-forward network, with a customisable number of hidden layers and neurons per layer. This is a significant deviation from the other approaches in that it does not calculate the suitability of an applicant as a straightforward weighted average.

Despite the differing structure of the neural network, the inputs and outputs are the same as that described in section 3.9.2 Role learning allocation model. The output layer still uses a sigmoid activation function, such that an output value greater than 0.5 indicates that the first applicant is more suitable, however the hidden layers may use a different activation function.

Similarly to the neural audition model, the complex network allocation engine uses the SGD training algorithm with the same configuration parameters described in section 3.9.1 Neural audition model, with the following additions;

- Hidden layers- the number of hidden layers in the feedforward network structure
- Neurons per layer- the number of neurons in each hidden layer
- Hidden layer activation function- the activation function used by the neurons in the hidden layers

The training schedule parameter defined in section 3.9.3 Session learning allocation model is also available.

Due to the complexity of this network, there is no guarantee (or even likelihood) that comparisons will always be consistent. That is, applicant A > B and B > C does not necessarily mean that A > C. Due to this inconsistency, normal sorting algorithms may produce inaccurate orderings or fail altogether, so I have designed and implemented an original sort algorithm called “Disagreement Sort”. This is described in Appendix I - Disagreement sort algorithm for those who are interested.

3.10 Automation of cast selection

In order to assess the balance of talent between casts, the cast selection process has been automated to emulate the user accepting the recommended selection. This automation was run using each of the implemented selection engines, with each year of CGS historical cast selection data (from 1993 to 2021).

The automation procedure, for a given year and selection engine, is as follows;

1. Convert the historical CGS cast selection data, for the chosen year, from the previous CE data file format to the new CARMEN ShowModel database format
2. Detect siblings in the cast by matching last names, and add these siblings as same cast sets*
3. Clear the alternative casts of the currently selected applicants
4. Initialise the chosen selection engine, and run the “balance alternative casts” operation, to reapply alternative casts to the selected cast

5. Measure the time taken to perform this operation
6. Calculate the statistical differences between talent in each alternative cast
7. Generate the output in TSV format

**It is important that the selection engine uses these same cast sets to keep siblings in the same cast, as this is a step which the human users would have performed manually when selecting cast, and it would not be a fair test if this requirement were skipped.*

The resulting dataset is analysed in section 4.1 Comparison of balance between casts, where each SAT based selection engine is compared to the heuristic engine and the human produced result.

3.11 Automation of role allocation

In order to assess the accuracy of the role allocation recommendations, and the spread of roles between applicants, the role allocation process has been automated to emulate the user accepting the recommended cast for each role. The automation procedure is broken up into 5 steps, which are described below.

This automation was run using each of the implemented allocation engines, with the odd years of the CGS historical casting data (from 1993 to 2021), while varying a number of engine parameters (as described in section 3.9 Implementation of the ANN model). The test data has been limited to the odd years to ensure a fair test on unseen data, as the even years have been used to train the models.

3.11.1 Step 1 - Enumerate models

Given the large number of engine parameters, which vary between engine implementations, the first step is to list all of the combinations of parameters and engine types which will be modelled.

In order to allow parallel processing between a number of computers, this step generates separate JSON files for each combination. The output files are then split into batches and moved between computers as required. This allows easy processing and data management, as the status is clear based on which folder the file is in, regardless of any intended or unintended closing of the process.

3.11.2 Step 2 - Train models

The next step is to train the models with historical casting data from the even years of CGS between 1994 and 2020 inclusive. The odd data is left unseen by the models so that it can be used in step 3 as a fair test.

The models are trained, with data from a given year or years, by the following procedure;

1. Read the model info from an input file generated by 3.11.1 Step 1 - Enumerate models
2. Initialise the model to its default values

3. If the model's engine type is one which will learn from user input,
    a. For each of the chosen years of training data,
        i. Convert the historical CGS casting data from the previous CE data file format to the new CARMEN ShowModel database format
        ii. Make a temporary copy of the converted ShowModel, with all role allocations cleared
        iii. Initialise the allocation engine with the provided engine parameters
        iv. For each role, in the simple role casting order*,
            1. Determine which applicants were available for this role at the time
            2. Re-allocate the originally cast applicants to the role
            3. Run the "user picked cast" operation of the allocation engine (which causes it to learn from which applicants were picked and which were not)
    b. Measure the time taken to complete this operation
4. Save the trained model to a new JSON file, and remove the input file to mark it as complete

**The simple role casting order is determined by the heuristic engine, and emulates the order roles were usually cast by the production team using the previous CE software.*

3.11.3 Step 3 - Test models

Now that the models are trained, the next step is to test them against unseen data. The CGS historical data from odd years between 1993 and 2021 inclusive can be used here as it was not used in the training step.

The models are tested, against data from a given year or years, by the following procedure;

1. Read the trained model from an input file generated by 3.11.2 Step 2 - Train models
2. For each of the chosen years of testing data,
    a. Convert the historical CGS casting data from the previous CE data file format to the new CARMEN ShowModel database format
    b. Measure an applicant summary* of the existing casting which was produced by human users
    c. Make a temporary copy of the converted ShowModel, with all role allocations cleared
    d. Initialise the allocation engine with the provided engine parameters and trained model
    e. For each role, in the simple role casting order (as per 3.11.2 Step 2 - Train models),
        i. Determine which applicants were available for this role at the time
        ii. For each pair of applicants (one which was previously allocated this role, and one which was not), use the allocation engine to predict which applicant was picked
        iii. Count the number of correct and incorrect comparison predictions
        iv. Re-allocate the originally cast applicants to the role
        v. Run the "user picked cast" operation of the allocation engine (which may cause additional learning** depending on the type of engine and parameters)
    f. Clear the role allocations in the temporary ShowModel, and reinitialise the engine
    g. For each role, in the recommended*** role casting order,

        i. Allocate the role to the recommended applicants, as provided by the engine
        ii. Compare the applicants picked for each role to those actually picked by the human users of the CGS production team
        iii. Count the casting results****, either as exact matches, or similar matches based on marks
        iv. Continue until the entire show has been cast automatically by the engine
    h. Measure an applicant summary* of the final casting which was chosen entirely by the engine
3. Save the comparison results, cast matching results, auto-cast applicant summary and human cast applicant summary, to a new JSON file, and remove the input file to mark it as complete

**The applicant summary counts how many roles, of each type, are allocated to each applicant.*

***The additional learning is allowed on the basis that if this were being done by a real user casting a real show, the engine would have the same opportunity to learn from the user casting the first role, then reapply that knowledge when recommending cast for subsequent roles.*

****The recommended casting order is determined by the allocation engine (see 3.7.3 Allocation engine) and is significantly more advanced than the simple casting order previously followed by the production team.*

*****The casting results measure, for each role, the number of applicants who were exactly matching those previous chosen, have all identical marks as those previously chosen, have identical marks for the required criteria, have all similar marks (within 5 marks out of 100) or have similar marks (within 5 marks out of 100) for the required criteria.*

3.11.4 Step 4 - Evaluate models

With the heavy lifting done in the previous step, step 4 combines the individual results into a complete dataset. Each tested model from step 3 is converted into a row* in each of the following output files;

1. Accuracy.csv, which contains
    a. The percentage accuracy of comparisons
    b. The percentage accuracy of casting results by each metric (see 3.11.3 footnote " **** ")
2. Distribution.csv, which contains
    a. The percentage of applicants which were allocated roles of each type
    b. The average number of roles received by each applicant (who received at least 1)
    c. The standard deviation in the number of roles received by each applicant (of at least 1)
    d. The cumulative probability of receiving a role based on an applicant's mark in the relevant criteria
3. Values.csv, which contains
    a. The final weights applied to each requirement in the suitability calculation
    b. The final weights applied to the overall ability in the suitability calculation

    c. The final costs applied to existing roles of each type in the suitability calculation
4. Training.csv, which contains
    a. The time taken to train the model
    b. The time taken to test the model
    c. The number of training operations performed on the model
    d. The number of times the weights were changed by a training operation

**Depending on the model type, some output rows are not relevant and will not be added. Some may also be duplicates because the result did not depend on all input parameters, which are then omitted. Any complex neural models which failed during training, that is, resulted in neural weights of +/- infinity, are also omitted.*

3.11.5 Step 5 - Aggregate models

The final step is to aggregate the raw data from step 4 into cross-tabulated CSV files which are easier to graph or analyse further. After choosing interactively which columns to aggregate across, the average value of the combined rows is calculated for each column and outputted to new files with the aggregated columns removed.

These aggregated files are then analysed in section 4.2 Comparison of recommendation accuracy to compare the accuracy of casting recommendations from each allocation engine to that of the heuristic allocation engine. Additionally the spread of roles between applicants is analysed and compared with both the heuristic allocation engine and human produced results.

4. Analysis

This section will set out my results from the testing described in section 3. Methodology. The majority of this work has been completed using Displayr, a cloud-based data analysis software package (Displayr, 2021), which is the source of any graphs or visual outputs which are otherwise unattributed.

4.1 Comparison of balance between casts

As outlined in section 3.8 Implementation of the SAT solving algorithm, a number of approaches have been implemented for the selection engine, responsible for ensuring an even balance of talent between alternative casts. The analysis here is comparing each of these engines to the heuristic selection engine baseline, and also to the human produced result from each year.

There are a number of metrics used to measure this balance of cast, with the results for each being described in the sub-sections below.

4.1.1 Mean difference

The mean difference metric is calculated as the average of the absolute differences between the average marks for each criteria, in each alternative cast. The full range of values calculated, for each combination of test year and engine, are included in Appendix J - Alternative cast balance results. The overall values for each engine, averaged across all years of test data, are shown below in Figure 13.

	Mean difference
Heuristic baseline	3.8 ↑
Human produced	3.0 ↑
Chunked pairs	1.2 ↓
Top pairs	1.9 ↓
Hybrid pairs	1.4 ↓
Three's a Crowd	2.0 ↓
Rank difference	4.5 ↑
Best pairs	2.5
NET	2.5

Engine by Mean difference
sample size = 1392; 95% confidence level

Figure 13 - Mean difference by engine type (lower is better)

These results show that on average, the pairs based approaches (chunked pairs, top pairs and hybrid pairs) are a significant improvement over both the heuristic baseline and the human produced result. It's

worth noting that in terms of engine grouping, best pairs is not considered a pairs based approach as it is much more similar to the rank difference engine in its underlying minimisation of the cost function.

The rank difference approach was the only engine which produced a larger mean difference than the human produced result, however this is unsurprising as the rank difference cost function looks only at ordering and not absolute marks.

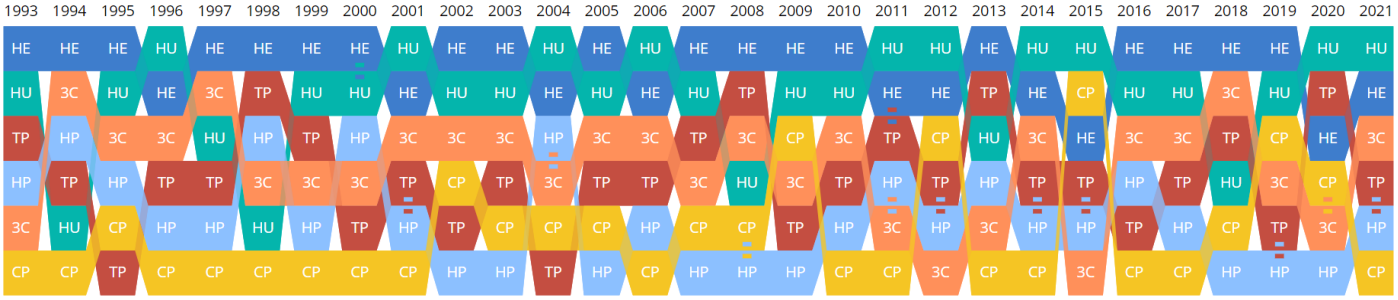


Figure 14 - Engine rankings by mean difference, excluding rank based (closer to the bottom is better)

The ranking of engines by year shows that the chunked pairs approach is most often the best (16 out of 29 years), followed by hybrid pairs (9 out of 29). Both are consistently in the top half, often sharing 1st and 2nd place between them.

Excluding the rank difference based approaches, for the reason stated above, the heuristic engine is most often the worst (19 out of 29 years). The human produced result is the worst case in the remaining 10 years, but shows a high degree of variability, sometimes reaching as high as the 2nd best approach for a given year. This is understandable as human efforts are likely to vary, and may have depended on factors such as who was present at the cast selection day, or how much attention was given to the cast balance, which is just one part of what the production team tries to achieve.

4.1.2 Mean difference (Top 5)

Additionally to the overall mean difference, the mean difference of the top 5 cast members for each criteria in each cast has been calculated (see Appendix J - Alternative cast balance results). This represents a realistic use case in that it is more valuable to have a balanced cast at the top of the talent pool, as these are the cast members who are most likely to be allocated lead roles. The overall values for each engine, averaged across all years of test data, are shown below in Figure 15.

Average	Mean difference (Top 5)
Heuristic baseline	5.2 ↑
Human produced	3.4
Chunked pairs	2.6 ↓
Top pairs	1.6 ↓
Hybrid pairs	1.6 ↓
Three's a crowd	3.3
Rank difference	6.9 ↑
Best pairs	1.7 ↓
NET	3.3

Engine by Mean difference (Top 5)
sample size = 1392; 95% confidence level

Figure 15 - Top 5 mean difference by engine type (lower is better)

These results show that the top pairs, hybrid pairs and best pairs approaches are the most effective at balancing the top talent of each criteria, with the chunked pairs and three's a crowd approaches achieving less valuable results. However once again, only the rank difference approach was worse than the human produced or heuristic baseline result.

Another important result here is that the difference between the human result and baseline heuristic is much greater for the top 5 metric than the overall metric. This is consistent with the human process implemented by the production team, which has only ever focussed on ensuring the top talent is balanced well.

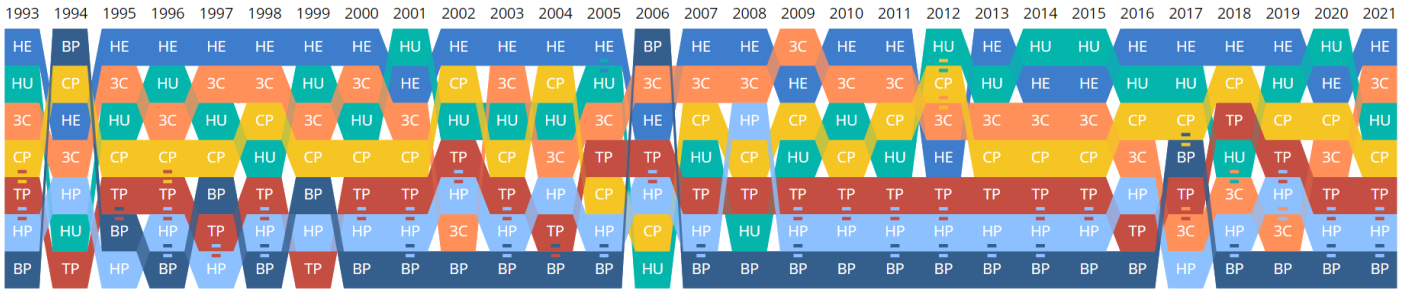


Figure 16 - Engine rankings by top 5 mean difference, excluding RD (closer to the bottom is better)

For the top 5 metric, the best pairs approach is most often the best with a staggering 23 out of 29 years. The hybrid pairs and top pairs are a close 2nd and 3rd, with the 3 of them almost exclusively appearing as the best 3, often tied.

4.1.3 Rank difference

The rank difference metric is a measure of the difference in the sum of the ranks in each cast. This is calculated as follows;

1. For each primary criteria and cast group,
    a. List the applicants in ascending order of their marks for the given criteria
    b. Assign the rank of 1 to the lowest mark, up to N for the highest mark, where N is the number of unique applicant marks (anyone with the same mark receives the same rank)
    c. Sum the ranks in each alternative cast, and find the absolute difference
2. Average the absolute rank differences across all primary criteria and cast groups

The full range of rank differences calculated, for each combination of test year and engine, are included in Appendix J - Alternative cast balance results. The overall values for each engine, averaged across all years of test data, are shown below in Figure 17.

Average	Rank difference
Heuristic baseline	23.4 ↑
Human produced	18.0 ▲
Chunked pairs	4.7 ↓
Top pairs	10.9 ↓
Hybrid pairs	7.4 ↓
Three's a crowd	10.7 ↓
Rank difference	30.6 ↑
Best pairs	12.4 ↓
NET	14.8

Engine by Rank difference
sample size = 1392; 95% confidence level

Figure 17 - Rank difference by engine type (lower is better)

These results show that on average, the chunked pairs approach is the most successful at minimising the rank difference. It also shows that all of the remaining SAT based approaches, except rank difference, perform better than the human produced result and significantly better than the heuristic approach. Despite the rank difference approach being specifically designed to minimise the rank difference, it appears to produce poor results on average. The possible reasons for this result are discussed in section 5.1 First hypothesis.

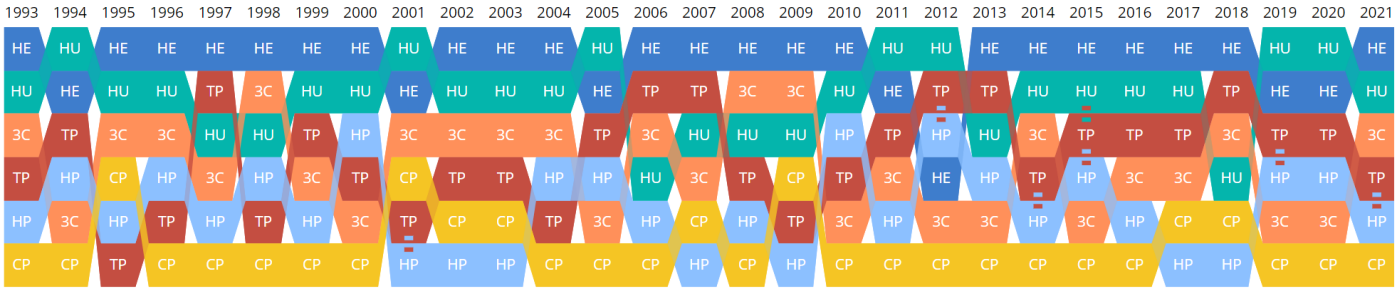


Figure 18 - Engine rankings by rank difference, excluding rank based (closer to the bottom is better)

The ranking of engines by year shows that the chunked pairs approach is most often the best (21 out of 29 years), followed by hybrid pairs (7 out of 29). This reaffirms the result from the mean difference metric.

Excluding the rank difference based approaches, the heuristic engine is most often the worst (22 out of 29 years) with the human produced result as the worst for the remaining 7 years. Again this reaffirms the result from the mean difference metric, where the human result shows a higher degree of variability.

4.1.4 Rank difference (Top 5)

Similarly to the mean difference top 5 metric, the rank difference of the top 5 cast members for each criteria in each cast has been calculated (see Appendix J - Alternative cast balance results). The overall values for each engine, averaged across all years of test data, are shown below in Figure 19.

Average	Rank difference (Top 5)
Heuristic baseline	6.2 ↑
Human produced	3.7
Chunked pairs	3.0 ↓
Top pairs	1.8 ↓
Hybrid pairs	1.8 ↓
Three's a crowd	4.0
Rank difference	7.3 ↑
Best pairs	1.9 ↓
NET	3.7

Engine by Rank difference (Top 5)
sample size = 1392; 95% confidence level

Figure 19 - Top 5 rank difference by engine type (lower is better)

These results show that the top pairs, hybrid pairs and best pairs approaches are the most effective at balancing the top talent of each criteria, but that the “three’s a crowd” approach is actually worse, on average, than the human produced result. Similarly to other results, the rank difference approach performs poorly and will be discounted from the ranking “bump” plot in Figure 20 below.

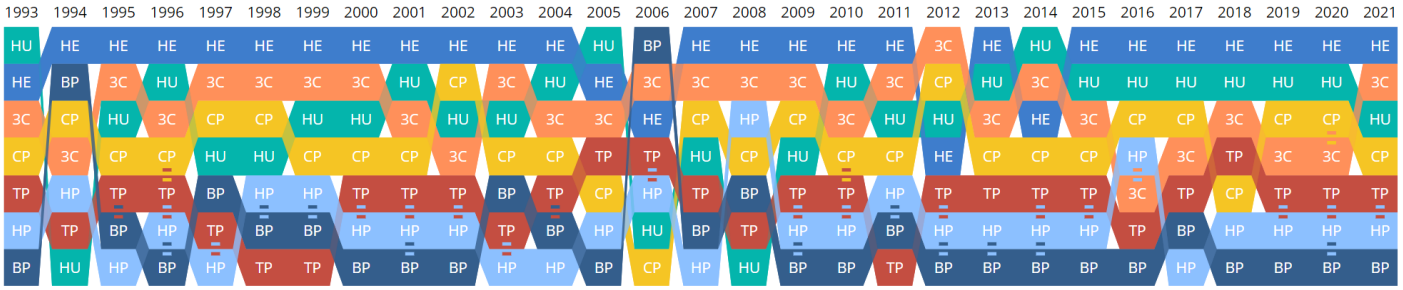


Figure 20 - Engine ranking by top 5 rank difference, excluding RD (closer to the bottom is better)

For the top 5 metric, the best pairs approach is most often the best (17 out of 29 years), but with a higher degree of variability than other approaches. It is however worth noting that many of the hybrid pairs results are tied best with best pairs and it would be equally valid to say that it was the best for 13 out of 29 years.

4.1.5 Time taken

Although the time taken does not affect the balance of the cast, it is an important factor in the real world use of these algorithms. As such, the full range of times measured, for each combination of test year and engine, are included in Appendix J - Alternative cast balance results. All times are measured in seconds.

Maximum	Seconds
Heuristic baseline	.0
Human produced	.0
Chunked pairs	3.8
Top pairs	.7
Hybrid pairs	22.4
Three's a crowd	.6
Rank difference	23.5
Best pairs	15.5

Engine by Total seconds
sample size = 1392; 95% confidence level

Figure 21 - Maximum time taken by engine type

Figure 21 above shows the maximum run times for each of the approaches, across all years, showing that the top pairs and “three’s a crowd” approaches are the fastest to execute.

Figure 22 below shows that although the rank difference has the slowest single time, it was only greater than 5.5 seconds for 6 out of 29 years, with 12 of the slowest 20 being the best pairs approach. These approaches are likely slow due to the rank difference cost function, which is common to both of them.

The remaining two slow tests were the hybrid pairs approach, which can be explained by its iterative nature of applying the chunked pairs approach.

	Seconds
RankDifferenceSatEngine_converted2018	23.5
HybridPairsSatEngine_converted2017	22.4
HybridPairsSatEngine_converted1993	20.3
BestPairsSatEngine_converted2018	15.5
RankDifferenceSatEngine_converted2005	13.4
BestPairsSatEngine_converted2008	12.2
BestPairsSatEngine_converted2019	12.0
RankDifferenceSatEngine_converted2016	10.8
BestPairsSatEngine_converted2007	10.1
BestPairsSatEngine_converted2020	8.6
BestPairsSatEngine_converted2017	7.9
BestPairsSatEngine_converted2000	7.5
RankDifferenceSatEngine_converted2021	7.4
BestPairsSatEngine_converted2016	7.2
BestPairsSatEngine_converted2015	6.0
BestPairsSatEngine_converted1993	5.7
BestPairsSatEngine_converted2013	5.7
RankDifferenceSatEngine_converted1998	5.7
RankDifferenceSatEngine_converted2015	5.6
BestPairsSatEngine_converted2021	5.5

Figure 22 - Slowest 20 test runs by engine type and year

4.2 Comparison of recommendation accuracy

As outlined in section 3.9 Implementation of the ANN model, a number of approaches, with various parameters, have been implemented for the allocation engine, responsible for providing recommendations of applicants while casting roles. The analysis here is comparing each of these engines to the heuristic allocation engine baseline. Additionally, an approach called the “ideal weights” engine has been included for comparison using the same weighted average which the heuristic uses, but with weights calculated externally by regression to match the entire 29 years of historical casting data. This represents the “best case” of a heuristic-style approach, but is only useful as a comparison tool rather than a real world approach.

The sections below firstly discuss the outcome of my proof of concept tests using ML.NET, and then the results from the various metrics of recommendation accuracy as described in section 3.11 Automation of role allocation.

4.2.1 Pointwise vs pairwise

Before implementing the testable models, a proof of concept was constructed with the machine learning package ML.NET to determine feasibility. The first scenario tested was pointwise, where the inputs to the

machine learning model were the marks of an applicant, and requirements of the role. The expected output used to train was whether or not an applicant was picked for that role.

This scenario was modelled as a classification task, with ML.NET configured to evaluate a variety of training algorithms for a total of 5 minutes, with 70,313 training data points from the even years of historical data. The output of this training evaluation is included in Figure 23 below.

```
---------------------------------------------------------------------------------------------------------------------
|                                                  Summary                                                          |
---------------------------------------------------------------------------------------------------------------------
|ML Task: Classification                                                                                            |
|Dataset: Casting_Pointwise(Train).csv                                                                              |
|Label : Picked                                                                                                     |
|Total experiment time : 244.71 Secs                                                                                |
|Total number of models explored: 154                                                                               |
---------------------------------------------------------------------------------------------------------------------
|                                            Top 5 models explored                                                  |
---------------------------------------------------------------------------------------------------------------------
|     Trainer                              Accuracy      AUC    AUPRC  F1-score  Duration #Iteration              |
|153  LightGbmBinary                         0.9223   0.8628   0.4538    0.3444       0.4         153              |
|151  FastForestBinary                       0.9213   0.8542   0.4400    0.2750       5.7         151              |
|145  FastForestBinary                       0.9212   0.8523   0.4329    0.2746       0.9         145              |
|94   LightGbmBinary                         0.9208   0.8633   0.4464    0.3624       0.5          94              |
|31   FastForestBinary                       0.9199   0.8513   0.4177    0.2483       0.7          31              |
---------------------------------------------------------------------------------------------------------------------
```

Figure 23 - Pointwise casting training in ML.NET

The results initially appeared promising, showing an overall accuracy of ~92%, which held up when the resulting best model was tested against every year of data. However, breaking down the accuracy by output value showed the true result.

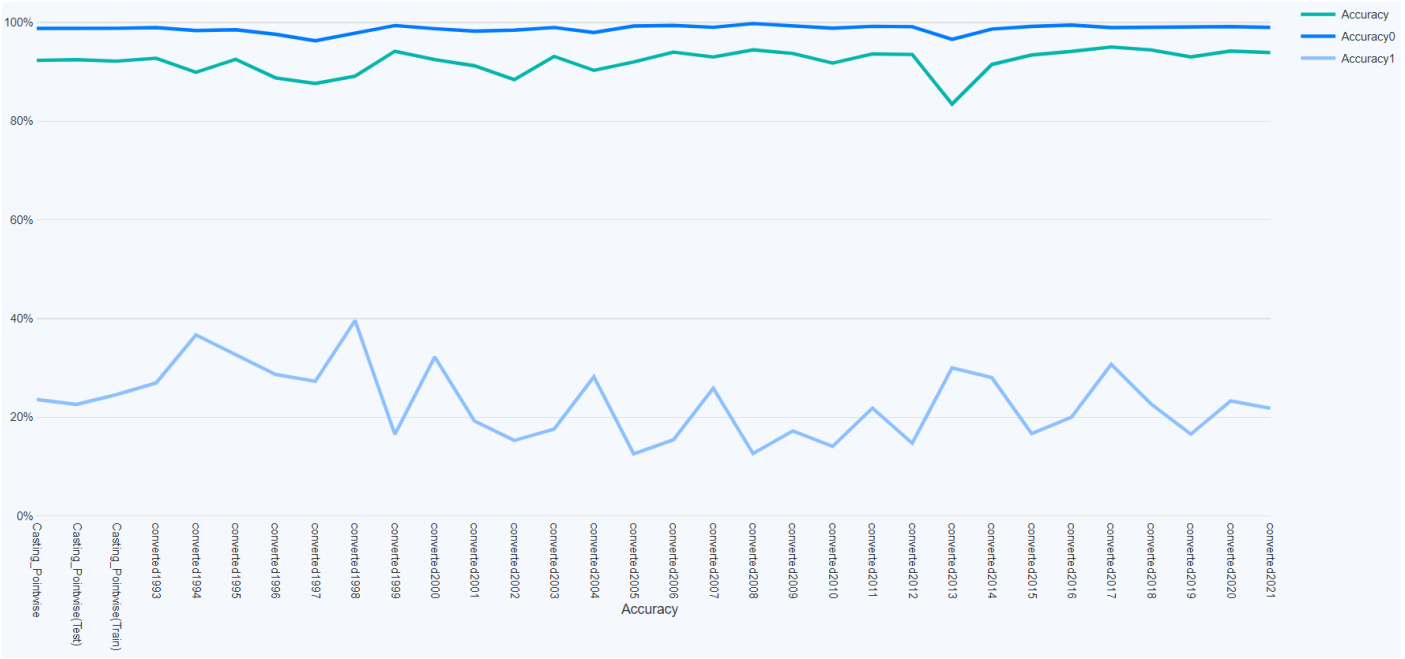


Figure 24 - Pointwise accuracy by year

While the overall accuracy looks fine, varying from year to year between 83% and 95%, the accuracy of predicting 0 (representing “not picked”) was almost 100%, with the accuracy of predicting 1 (representing

“picked”) between only 13% and 40%. The model has more or less learned to never pick anyone, and you’ll be correct most of the time. On reflection this makes sense, as for every role in which 1 applicant is picked, every other applicant will be not picked.

The second scenario tested with ML.NET was pairwise, where the inputs to the machine learning model were the marks of two applicants, and requirements of the role. The expected output used to train was whether the first applicant was more suitable than the second applicant for the given role.

This scenario was also modelled as a classification task, with ML.NET configured to evaluate a variety of training algorithms for a total of 10 minutes, with 244,218 training data points from the even years of historical data. The output of this training evaluation is included in Figure 25 below.

** The increased time of 10 minutes was recommended by ML.NET due to the size of the test data set*

```
--------------------------------------------------------------------------------------------------------------------
|                                                  Summary                                                         |
--------------------------------------------------------------------------------------------------------------------
|ML Task: Classification                                                                                           |
|Dataset: Casting_Pairwise(Train).csv                                                                              |
|Label : A_BetterThan_B                                                                                            |
|Total experiment time : 316.43 Secs                                                                               |
|Total number of models explored: 60                                                                               |
--------------------------------------------------------------------------------------------------------------------
|                                            Top 5 models explored                                                 |
--------------------------------------------------------------------------------------------------------------------
|     Trainer                              Accuracy      AUC    AUPRC  F1-score  Duration #Iteration             |
|43   FastTreeBinary                         0.8246   0.9027   0.9025    0.8256       0.8         43             |
|8    FastTreeBinary                         0.8203   0.8953   0.8946    0.8213       0.9          8             |
|9    FastTreeBinary                         0.8173   0.8904   0.8857    0.8177     272.2          9             |
|4    FastTreeBinary                         0.8020   0.8802   0.8770    0.8031       0.5          4             |
|27   LightGbmBinary                         0.7982   0.8785   0.8761    0.8067       0.3         27             |
--------------------------------------------------------------------------------------------------------------------
```

Figure 25 - Pairwise casting training in ML.NET

The training results show an overall accuracy of ~82% and are once again confirmed by evaluating this model against each year of casting data, as shown in Figure 26 below. The accuracy varied from year to year between 73% and 94%, but with the accuracy of individual value predictions never going below 72%.

Although this has a lower overall stated accuracy than pointwise, the pairwise model proves to be much more balanced between predictions by output value. This inherently makes sense in the domain of theatrical role casting, as there are often multiple applicants who are suitable in their own right for a given role, but the one who will be ultimately chosen depends on who else is available at the time.

Based on this proof of concept, all neural network models in CARMEN have been implemented using the pairwise approach.

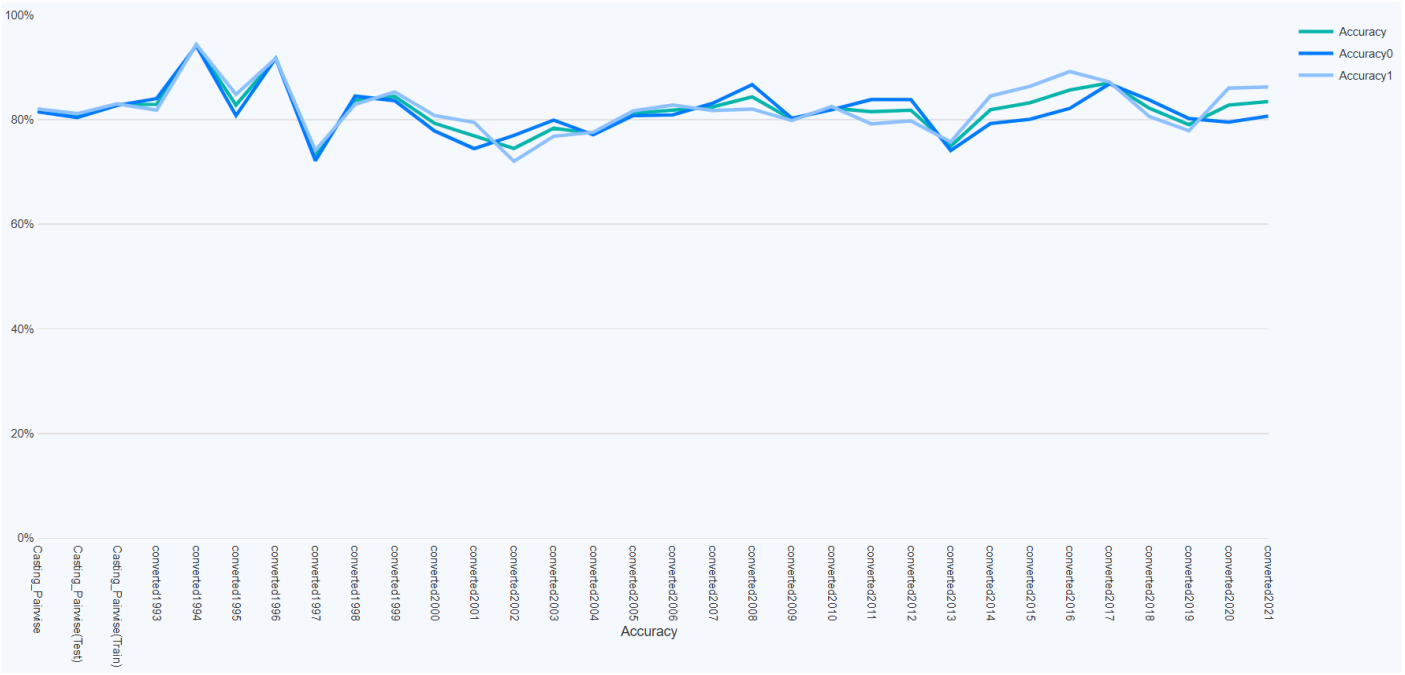


Figure 26 - Pairwise accuracy by year

4.2.2 Comparison accuracy

The comparison accuracy metric is a measure of how accurate each model is at predicting which out of two applicants is more suitable for a given role. It is calculated by the procedure described in point 3a of section 3.11.2 Step 2 - Train models, that is by identifying the applicants picked and not picked for each role, given who would have been available at the time. These lists are paired in every combination, based on the assumption that every applicant picked for a role must be more suitable than every applicant not picked for that role, and vice versa. For each of these data points, across all the odd years of historical data, the model being tested is used to predict which applicant is more suitable. The comparison accuracy is then calculated as the number of correct predictions out of the total predictions, and has been done for a total of 50,992 combinations of engine type and parameters (full data available on request).

Accuracy	Minimum	Average	Maximum	Sample Size
Complex network	.365	.517 ↓	.930	1,437
Heuristic	.867	.867	.867	6
Role learning	.730	.858 ↑	.876	13,640
Session learning	.730	.863 ↑	.876	35,873
Ideal weights	.868	.869 ↑	.870	36

Figure 27 - Comparison accuracy by engine type

The heuristic baseline shows a consistently high accuracy of 86.7%, which is in line with the results of the POC tests. The high correlation is unsurprising given that this is the exact approach used in the previously CE software which was used during these years of casting and therefore only differs where the human users chose specific applicants rather than accepting the recommendation.

The ideal weight engines show a slight increase ~0.2% over the heuristic, consistent with its design as the “best case” of a heuristic-style approach.

The role learning and session learning approaches show an average accuracy 0.9% and 0.4% lower respectively than the heuristic, but with significantly higher variation across the test cases. This suggests that the quality of the approach depends upon the engine parameters used, which are examined in more detail below. An encouraging observation here is that at least some neural models show a 0.9% improvement over the heuristic.

The complex network results vary between extremes of 37% and 93%. Similarly to the other neural network models, this suggests that the quality of the model depends upon engine parameters, however unlike the others the complex network models can fail entirely, as any result below 50% is no better than guessing.

4.2.2.1 Suitability calculation parameters

The first engine parameter examined is how the overall ability is weighted in the suitability calculation. This can either be by a single overall ability weight, a different overall ability weight per requirement, or no weighting at all. Both heuristic and complex network only use the common overall weight, and are therefore excluded from Figure 28 below.

Average		Accuracy
RL	Common Overall	.854 ↓
	No Overall	.859 ↑
	Vector Overall	.859 ↑
SL	Common Overall	.859 ↓
	No Overall	.865 ↑
	Vector Overall	.865 ↑
IW	Common Overall	.870 ↑
	No Overall	.868 ↓
	Vector Overall	.870 ↑

Figure 28 - Comparison accuracy by how overall ability is weighted

From this we can see that the role and session learning models work best with no overall weight or a vector of different weights, whereas the ideal weight model works best with a vector or common. Therefore vector is the best option for this parameter, and will be used as the default setting in the final application.

The next parameter is how costs for existing roles are subtracted in the suitability calculation. They can either be subtracted from the requirement suitability before the weighted average of requirements is calculated, or from the final suitability afterwards. This is relevant to all engine types except heuristic, which always subtracts from the requirement.

Average		Accuracy
CN	Final	.634 ↑
	Requirement	.505 ↓
RL	Final	.857 ↓
	Requirement	.858 ↑
SL	Final	.863
	Requirement	.863
IW	Final	.869
	Requirement	.869

Figure 29 - Comparison accuracy by how costs are subtracted for existing roles

In Figure 29 above we can see that complex networks perform significantly better (13.1%) when costs are subtracted from the final suitability, but role learning is slightly better (0.1%) when subtracted from requirements. Session learning and ideal weights show no change.

Average		Accuracy
CN	GeometricMean	.489 ↓
	Whole Roles	.530 ↑
HE	GeometricMean	.867
	Whole Roles	.867
RL	GeometricMean	.858
	Whole Roles	.858
SL	GeometricMean	.863
	Whole Roles	.863
IW	GeometricMean	.869
	Whole Roles	.869

Figure 30 - Comparison accuracy by how existing roles are counted

Following on from how existing role costs are subtracted is how the roles are counted in the first place. These can either be counted as whole roles (eg. a Singer/Actor role counts as 1 singing role and 1 acting role) or as a geometric mean (a Singer/Actor role counts at ~0.7 singing roles and ~0.7 acting roles). Figure 30 above shows that Complex networks perform on average 4.1% better by counting whole roles, but all other engine types show no difference.

4.2.2.2 Neural learning parameters

The next set of parameters are those used by the neural network training algorithm, such as the loss function. Figure 31 below shows that the complex network models function substantially better with the binary cross-entropy loss function despite its lackluster performance for other engine types.

Average	CN	RL	SL(ER)	SL(RS)	SL(PS)
Binary X	.732 ↑	.844 ↓	.842 ↓	.862 ↓	.871 ↑
P > 0.9	.464 ↓	.863 ↑	.862 ↑	.867 ↑	.871
P > 0.8	.485 ↓	.862 ↑	.861 ↑	.867 ↑	.871
P > 0.7	.558 ↑	.862 ↑	.861 ↑	.867 ↑	.871
P > 0.6	.516	.861 ↑	.860 ↑	.867 ↑	.871
P > 0.5	.495 ↓	.861 ↑	.860 ↑	.867 ↑	.871
MAE	.503	.854 ↓	.853 ↓	.866 ↑	.871
MSE	.530	.854 ↓	.853 ↓	.866 ↑	.871

Figure 31 - Comparison accuracy by loss function

The results also show a mid-range peak in the classification loss functions (P > threshold) at a threshold value of 0.7, reducing in accuracy towards either extreme. For the purposes of default engine parameters, binary cross-entropy will be used for complex networks.

The role learning and session learning (every role) models perform best with the classification loss function, increasing their accuracy as the threshold is increased. Session learning with role stockpiling works equally well for all classification loss thresholds, as does session learning (per session), however the interesting result for the latter is that the accuracy does not vary in regard to loss function, with even binary cross-entropy producing the same result. As the overall winner, the “P > 0.9” classification function will be used as the default value for non-complex neural models.

Average	CN	RL	SL(ER)	SL(RS)	SL(PS)
0.0001	.592 ↑	.868 ↑	.868 ↑	.867 ↑	.870 ↓
0.0005	.746 ↑	.866 ↑	.865 ↑	.867 ↑	.870 ↓
0.001	NaN	.864 ↑	.863 ↑	.867 ↑	.871 ↓
0.005	.455 ↓	.857 ↓	.855 ↓	.867 ↑	.871 ↑
0.01	.484 ↓	.853 ↓	.851 ↓	.867 ↑	.872 ↑
0.05	.508 ↓	.839 ↓	.837 ↓	.863 ↓	.872 ↑

Figure 32 - Comparison accuracy by neural learning rate

For most neural models, a slower learning rate results in a more accurate model, with both the role learning and session learning (every role) models continuing to improve all the way to the slowest learning rate tested. The complex network trend is less well defined, due to a much smaller and varied sample than the role and session learning models, but it is clear that slower is better in general. Session learning with role stockpiling works better at slower learning rates but plateaus below 0.01. Only the session learning (per session) models performed worse with slower learning rates, but at only 0.2% difference between extremes, this is negligible.

Average	CN	RL	SL(ER)	SL(RS)	SL(PS)
1	.402 ↓	.864 ↑	.862 ↑	.866	.870 ↓
10	.506 ↓	.858	.856	.866	.871
100	.523 ↑	.852 ↓	.851 ↓	.866	.872 ↑

Figure 33 - Comparison accuracy by max iterations of neural network training

In a near identical result, slower training (in this case by a lower number of iterations) yields better results for role learning and session learning (every role) models, and having no effect on session learning with role stockpiling. Session learning (per session) once again prefers faster training, with a 0.2% decrease in accuracy from the highest iterations compared to the lowest, which is negligible.

However, unlike the learning rate result, complex network models perform better with more training iterations. This is understandable given the significantly higher complexity of the complex networks, as it will take more iterations of training to align the multiple neurons into a usable prediction model, compared to the single neuron used by other engine types.

Session learning:

Average Row Sample Size	Accuracy
Every Role	.856 ↓ 15,364.000
Stockpile Roles	.866 ↑ 15,381.000
Per Session	.871 ↑ 5,128.000

Complex network:

Average Row Sample Size		Accuracy
Final	Every Role	.446 ↓ 10.000
	Stockpile Roles	.500 6.000
	Per Session	.657 ↑ 116.000
Requirement	Every Role	.639 ↑ 288.000
	Stockpile Roles	.635 ↑ 108.000
	Per Session	.447 ↓ 909.000

Figure 34 - Comparison accuracy by the type of neural network training

The most accurate session learning models train once per session, followed by role stockpiling and then training every role. The same pattern applies for complex networks when existing role costs are subtracted from the final suitability, but the inverse is true when subtracting from the requirement suitabilities. This option is not available for role learning, which always trains once per role using details specific to that role.

The “reload weights” parameter was introduced on the premise that resetting the network after each operation would stop small changes from being amplified when they weren’t significant.

Session learning:

Average		Accuracy
Every Role	Always	.854 ↓
	On Change	.854 ↓
	When Refused	.862
Stockpile Roles	Always	.866 ↑
	On Change	.866 ↑
	When Refused	.867 ↑

Role learning:

Average	Accuracy
Always	.856 ↓
On Change	.856 ↓
When Refused	.861 ↑

Figure 35 - Comparison accuracy by how often the neural network weights are reloaded

Figure 35 above shows that this may have an element of truth when stockpiling roles, which had no variation in accuracy, but in all other cases only reloading when specifically refused by the user found a significantly better result. As such, reloading only when refused will be used as the default setting.

Average	Binary-X	P > 0.9	P > 0.8	P > 0.7	P > 0.6	P > 0.5	MAE	MSE
ExponentialLu	.789	.397 ↓	.446 ↓	.563	.393 ↓	.418 ↓	.396 ↓	.392 ↓
LeakyReLu	NaN	.682 ↑	.530 ↑	.604 ↑	.544 ↑	.601 ↑	.430 ↓	.651 ↑
ReLu	NaN	.500 ↑	.624 ↑	.548	.549	.570 ↑	.585 ↑	.589
Sigmoid	.652 ↓	.500 ↑	.500 ↑	.500 ↓	.500	.500	.500	.500 ↓
Tanh	.783 ↑	.400 ↓	.407 ↓	.515	.500	.481	.529	.498 ↓

Figure 36 - Comparison accuracy by the activation and loss functions used in the complex network

The most effective activation function used for the hidden layers in a complex network was found to be dependent on the loss function used for training. From earlier results, the recommended loss function for complex networks was binary cross-entropy, which is most accurate when used with the exponential linear unit, or tanh activation functions with 78.9% and 78.3% accuracy respectively. The third best result came from the P > 0.9 loss function when used with the leaky rectified linear unit activation function, which also worked well on average with all classification threshold loss functions.

Average	1 Layer	2 Layers	3 Layers
21	.606	.511 ↑	.474 ↑
42	.617 ↑	.450 ↓	.457 ↓
84	.524 ↓	.456 ↓	.463

Figure 37 - Comparison accuracy by the layers and neurons in the complex network

The final parameter to evaluate is the number of hidden layers and neurons per layer which form the neural structure of the complex network. The complex network approach is substantially different to all other approaches in that the suitability is not calculated by a simple weighted average, which is equivalent to a single neuron. The complex network is instead made up of additional layers and neurons which may or may

not improve the overall accuracy. The results in Figure 37 above show that the less complex the model, ie. the less number of hidden layers and the less neurons per layer, the more accurate the predictions will be. This is likely due to overfitting of the test data, and provides a good indication that the simplicity of the other approaches is justified for this task. The one caveat is that a single hidden layer of 21 neurons performs worse than that of 42 neurons, but this is reasonable given that the network has 42 inputs, so put simply, a total of 21 neurons is not enough to capture the full data available from 42 inputs.

4.2.2.3 Model evaluation by comparisons

Now that the engine parameters have been analysed and default values chosen (see section 4.2.4 Default engine parameters), the models which would have been made by these defaults and reasonable alternatives have been compared to the heuristic baseline. For the purposes of this comparison, I have included all 3 training schedules for the session learning engine, as these are effectively different approaches, but had to exclude the 3 possible complex neural models due to a lack of matching data. The full table of 42 unique models (including the 2 heuristic baseline) can be found in Appendix K - Recommendation accuracy results, but is summarised in Figure 38 below.

#	Model specification	Accuracy
1	Session learning (every role), count whole roles	87.21%
5	Role learning, count whole roles, subtract from requirement	87.20%
7	Session learning (every role), count geometric roles	87.20%
11	Role learning, count geometric roles, subtract from requirement	87.20%
13	Role learning, count whole roles	87.19%
17	Session learning (per session), count whole roles	87.19%
21	Session learning (per session), count geometric roles	87.18%
25	Session learning (stockpile roles), count geometric roles	87.03%
29	Session learning (stockpile roles), count whole roles	87.02%
33	Ideal weights, vector overall weights	86.98%
37	Ideal weights, no overall weights	86.79%
41	Heuristic baseline	86.69%

Figure 38 - Summary of models by comparison accuracy

Of the 32 neural models which use the default or reasonable alternative values, all of them have a higher accuracy than the 2 heuristic models, and the 8 ideal weight models. The top 16 models, which are all based on learning per role, have an accuracy of 87.2%, a 0.5% improvement over the baseline heuristic, as do the next 8 models which learn by session. The role stockpiling models are ranked 25th-32nd with an

accuracy of 87.0%, still 0.3% higher than the heuristic. For comparison, the ideal weight models, which are supposed to be the best case of the heuristic, only beat the baseline by 0.1-0.3%.

4.2.3 Auto-casting accuracy

The auto-casting accuracy metrics are a measure of how accurately each model can cast an entire show autonomously. They are calculated by the procedure described in point 2g of section 3.11.3 Step 3 - Test models, that is by clearing the casting of a past show and recasting each role in the recommended casting order, accepting the recommended allocation each time. The automatic casting is then compared to the previous historical casting and measured for accuracy in the following categories;

1. Exact applicant- the percentage of applicants who were previous cast in each role
2. Same marks- the percentage of applicants who have the same marks as those previously cast
3. Same relevant marks- the percentage of applicants who have the same mark in the criteria required by the role
4. Similar marks- the percentage of applicants whose marks are within 5 (out of 100) of the previous cast marks
5. Similar relevant marks- the percentage of applicants whose marks in the criteria required by the role are within 5 (out of 100)

These metrics have been calculated for a total of 50,992 combinations of engine type and parameters (full data available on request).

As can be seen in Figure 39 below, casting accuracy percentages are significantly lower than comparison accuracies. The primary reason for this is that unlike comparison accuracies, the recommended cast are allocated to the roles, rather than re-casting the previous cast, effectively meaning that errors are propagated over the course of the show. This metric is also much more susceptible to differences in casting order.

Average Maximum Row Sample Size	Exact Applicant	Same Marks	Same Relevant Marks	Similar Marks	Similar Relevant Marks
Complex network	.049 ↓ .142 1,437.000	.050 ↓ .142 1,437.000	.107 ↓ .215 1,437.000	.051 ↓ .142 1,437.000	.157 ↓ .282 1,437.000
Heuristic	.247 .259 6.000	.248 .260 6.000	.419 .435 6.000	.251 .262 6.000	.510 .524 6.000
Role learning	.249 ↑ .264 13,640.000	.250 ↑ .264 13,640.000	.421 ↑ .441 13,640.000	.252 ↑ .267 13,640.000	.510 ↑ .529 13,640.000
Session learning	.248 ↑ .265 35,873.000	.249 ↑ .266 35,873.000	.419 ↑ .443 35,873.000	.251 ↑ .268 35,873.000	.509 ↑ .531 35,873.000
Ideal weights	.250 ↑ .263 36.000	.251 ↑ .263 36.000	.422 ↑ .436 36.000	.254 ↑ .265 36.000	.507 ↑ .520 36.000

Figure 39 - Casting accuracy by engine type

Similarly to the comparison accuracy, the complex network models perform poorly, with even the maximum accuracies approximately half that of the heuristic, and as such are discounted from the remainder of this section. Also similarly to comparison accuracy, the ideal weight models are a slight improvement (~0.3%) on the baseline heuristic for all but the similar relevant marks metric. However the interesting result here is that both the role learning and session learning models are on average a slight improvement over the heuristic (~0.1%) with maximums showing that there are at least some of each which are 0.5% more accurate.

4.2.3.1 Engine parameters

Figure 40 below shows that for role and session learning, both vector and no overall weight have significant improvement over common, with ideal weight models having the best result when used with vector over weights. This reaffirms the result from section 4.2.2 Comparison accuracy.

Average		Exact Applicant	Same Marks	Same Relevant Marks	Similar Marks	Similar Relevant Marks
RL	Common Overall	.244 ↑	.245 ↑	.412 ↑	.247 ↑	.509 ↑
	No Overall	.251 ↑	.252 ↑	.424 ↑	.254 ↑	.511 ↑
	Vector Overall	.251 ↑	.252 ↑	.424 ↑	.254 ↑	.511 ↑
SL	Common Overall	.242 ↓	.242 ↓	.409 ↓	.245 ↓	.506 ↑
	No Overall	.251 ↑	.252 ↑	.424 ↑	.254 ↑	.510 ↑
	Vector Overall	.251 ↑	.252 ↑	.424 ↑	.254 ↑	.510 ↑
IW	Common Overall	.250 ↑	.250 ↑	.422	.253 ↑	.508
	No Overall	.250 ↑	.250 ↑	.421	.253 ↑	.506
	Vector Overall	.252 ↑	.253 ↑	.423 ↑	.255 ↑	.508

Figure 40 - Casting accuracy by how overall ability is weighted

The method of how existing role costs are subtracted, and how the roles are counted, both show no impact to the casting accuracy metrics. Neither are there any significant differences by loss function, learning rate or training iterations.

Average Row Sample Size	Exact Applicant	Same Marks	Same Relevant Marks	Similar Marks	Similar Relevant Marks
Every Role	.246 ↓ 15,364.000	.246 ↓ 15,364.000	.416 ↓ 15,364.000	.249 ↓ 15,364.000	.508 ↓ 15,364.000
Stockpile Roles	.250 ↑ 15,381.000	.250 ↑ 15,381.000	.422 ↑ 15,381.000	.253 ↑ 15,381.000	.510 ↑ 15,381.000
Per Session	.249 ↑ 5,128.000	.250 ↑ 5,128.000	.421 ↑ 5,128.000	.253 ↑ 5,128.000	.510 ↑ 5,128.000

Figure 41 - Casting accuracy by type of neural network training

As was the case with the comparison accuracy metric, training session learning models once per session was significantly more effective than every role, however it's worth noting that the casting accuracy metrics show “stockpiling roles” as being equally as good, rather than halfway between “per session” and “every role”.

An additional variable in the casting metrics is the order in which the roles are cast. As alluded to at the start of this section, the order the roles are cast has a significant impact on the accuracy, as can be seen below in Figure 42.

Average		Exact Applicant	Same Marks	Same Relevant Marks	Similar Marks	Similar Relevant Marks
HE	IdealByBracket	.258	.259	.435	.262	.524
	IdealWholeShow	.251	.252	.433	.255	.522
	Simple	.233	.234	.390	.236	.486
RL	IdealByBracket	.256↑	.257↑	.434↑	.259↑	.522↑
	IdealWholeShow	.255↑	.256↑	.432↑	.258↑	.519↑
	Simple	.233↓	.234↓	.392↓	.237↓	.487↓
SL	IdealByBracket	.256↑	.256↑	.433↑	.259↑	.522↑
	IdealWholeShow	.254↑	.255↑	.431↑	.258↑	.518↑
	Simple	.233↓	.234↓	.392↓	.237↓	.487↓
IW	IdealByBracket	.260↑	.260↑	.435↑	.263↑	.519↑
	IdealWholeShow	.254↑	.254↑	.432↑	.257↑	.515↑
	Simple	.238↓	.239↓	.397↓	.241↓	.488↓

Figure 42 - Casting accuracy by the role order used to auto-cast

The ideal by bracket and ideal whole show orders improve the result by ~2% over the simple casting order, when comparing exact applicant and all marks, or ~4% when comparing only relevant marks. This is true for both the heuristic style and neural network models (excluding complex networks). Between the two versions of ideal order, ideal by bracket comes out on top across the board, though it is worth noting that the differences vary between engine types with heuristic style engines ~0.7% more accurate but neural engines only slightly (~0.2%) more accurate.

Although not directly related to my hypotheses, there is one further metric related to the auto-casting order which is worth analysing. Given the show has been automatically cast, there is no guarantee that every role is filled, as there could be scheduling conflicts on section edges which would normally be fixed by the user. Ideally you’d want the chosen casting order to get as close as possible to a 100% fully cast show, even if working autonomously. Therefore this metric of “roles fully cast” has been calculated as the number of roles fully cast out of the total number of roles in the show, and is shown in Figure 43 below.

Average		Roles Fully Cast
HE	IdealByBracket	.995
	IdealWholeShow	.991
	Simple	.993
RL	IdealByBracket	.995↑
	IdealWholeShow	.991↓
	Simple	.993↑
SL	IdealByBracket	.995↑
	IdealWholeShow	.991↓
	Simple	.993↑
IW	IdealByBracket	.995↑
	IdealWholeShow	.991↓
	Simple	.993↓

Figure 43 - Auto-casting completion by the role order used to auto-cast

This shows that the ideal by bracket order is the most effective at fully casting roles autonomously, reaffirming its position as the default recommended casting order.

4.2.3.2 Model evaluation by casting

In order to compare the casting accuracy of each approach to the heuristic, the list of models has been limited to those using the default parameters or reasonable alternatives listed in section 4.2.4 Default engine parameters, however all 3 training schedules for the session learning engine have been included, as these are effectively different approaches. Due to a missing data point, one of the role learning models is unavailable, leaving 39 models to compare to the 2 heuristic baseline models. These are summarised in Figure 44 below, with the full table found in Appendix K - Recommendation accuracy results.

Of the 31 neural models which use the default or reasonable alternative values, all of them have a higher casting accuracy, in at least one casting metric, than the best heuristic model in that metric. Although the heuristic model is ranked first in the “Similar relevant marks” metric, this is the exception to the rule as it is beaten by 9 neural models in “Exact applicants”, “Same marks” and “Similar marks”, and by all 31 in “Same relevant marks”. This indicates that the 52.4% similar relevant marks accuracy of the heuristic is an outlier, as even the ideal weight models could not match it, despite improving on the heuristic in every other category.

Excluding the similar relative marks outlier, the models which learn by role and count geometrically are ~0.2% more accurate than the heuristic across the board and are only beaten by the ideal weight models. This is consistent with the results in section 4.2.2 Comparison accuracy.

#	Model specification	Exact App.	Same Marks	Same Rel. Marks	Sim. Marks	Sim. Rel. Marks
1	Ideal weights, vector overall weights	**26.3%**	**26.3%**	**43.6%**	**26.5%**	52.0%
5	Session learning (every role), count geometric roles	**26.0-26.1%**	**26.1-26.2%**	**43.7%**	**26.4%**	52.2%
9	Role learning, count geometric roles* **vector overall weights, subtract from req'ment not tested*	**26.0%**	**26.0-26.1%**	**43.6-43.7%**	**26.3-26.4%**	52.1-52.2%
12	Heuristic baseline, count whole roles	25.9%	26.0%	43.4%	26.2%	52.4%
13	Session learning (per session), count geometric roles	25.8-25.9%	25.9-26.0%	**43.6-43.7%**	**26.2-26.3%**	52.1-52.2%
17	Session learning (every role), count whole roles	25.8%	25.8%	**43.7%**	26.1%	52.3%
21	Role learning, subtract from req'ment, count whole roles	25.8%	25.8%	**43.7%**	26.1%	52.3%
23	Heuristic baseline, count geometric roles	25.8%	25.8%	43.5%	26.1%	52.3%
24	Ideal weights, no overall weights	25.9%	25.9%	43.4%	26.1%	51.8%
28	Session learning (role stockpiling), count geometric roles	25.6%	25.7%	**43.7%**	25.9%	52.1%
32	Session learning (per session), count whole roles	25.6%	25.6%	**43.6%**	25.9%	52.2%
36	Role learning, subtract from final, count whole roles	25.6%	25.6%	**43.6%**	25.9%	52.2%
38	Session learning (role stockpiling), count whole roles	25.3-25.4%	25.4%	**43.6-43.7%**	25.6%	52.1-52.2%

Figure 44 - Summary of models by casting accuracy

4.2.4 Default engine parameters

Based on the results of sections 4.2.2 Comparison accuracy and 4.2.3 Auto-casting accuracy, the default engine parameters have been set as per Figure 45 below.

Engine Parameter	Relevant Engines	Default Setting	Reasonable Alternative
Overall Weight Type	IW, RL, SL	Vector of different weights	No overall weight
Subtract Costs From	IW, RL, SL	Final	Requirement
	CN	Final	
Count Roles By	HE, IW, RL, SL	Geometric mean	Whole roles
	CN	Whole roles	
Loss Function	RL, SL	P > 0.9	
	CN	Binary cross-entropy	P > 0.7
Learning Rate	RL, SL, CN	0.0001 (lowest tested)	
Training Iterations	RL, SL	1	
	CN	100	
Training Schedule	SL, CN	Once per session	
Reload Weights	RL, SL	Only reload when refused	
Activation Function	CN (Binary-X)	Exponential linear unit	Tanh
	CN (P > 0.7)	Leaky rectified linear unit	
Hidden Layers	CN	1 layer	
Layer Neurons	CN	42 neurons (= number of inputs)	
Casting Order*	HE, IW, RL, SL, CN	Ideal by bracket	

Figure 45 - Default engine parameters

**Casting order is only relevant to the auto-casting accuracy metric analysed in section 4.2.3 Auto-casting accuracy and not to the comparison accuracy metric in section 4.2.2 Comparison accuracy*

4.3 Comparison of role distribution

The final section of numerical analysis measures how evenly distributed the roles are between available applicants. This will compare each of the implemented approaches outlined in section 3.9 Implementation of the ANN model to the heuristic baseline and human result.

The spread of roles across the cast is measured by the percentage of applicants which receive a role of a specific type (Singing, Acting, Dancing) or any role at all. It is measured against the final role allocations of the auto-casting procedure described in point 2g of section 3.11.3 Step 3 - Test models, that is clearing the casting of a past show and recasting each role in the recommended casting order, accepting the recommended allocation each time. This metric has been calculated for each of the 49,557 tested models (full data available on request). This dataset excludes the 1,437 complex network models, as their accuracy was found to vary wildly in section 4.2.2 Comparison accuracy, but includes 2 additional data points representing the previously human-picked casting.

Average Maximum Row Sample Size	Singing Roles	Acting Roles	Dancing Roles	Any Roles
Heuristic	.458 .480 6.000	.287 .300 6.000	.406 .419 6.000	.663 .674 6.000
Human picked	.501 ▲ .501 2.000	.345 ▲ .345 2.000	.431 ▲ .431 2.000	.731 ▲ .731 2.000
Role learning	.454 ▲ .505 13,640.000	.291 ▼ .316 13,640.000	.397 ▼ .437 13,640.000	.668 ▲ .686 13,640.000
Session learning	.453 ▼ .509 35,873.000	.292 ▲ .327 35,873.000	.397 ▲ .479 35,873.000	.667 ▼ .690 35,873.000
Ideal weights	.458 .477 36.000	.286 ▼ .298 36.000	.397 .415 36.000	.673 ▲ .683 36.000

Figure 46 - Role spread by engine type

Figure 46 above shows that historically speaking the task of evenly distributing the roles was performed manually by the user, as the human result has 1-6% higher role spread than the best heuristic across the board. While the neural engine types have lower role spread on average, the maximum values indicate that there are some models which improve the role spread over the heuristic and even match the human result in some categories. The analysis below is focussed on which engine parameters produce these models.

Investigating how the role spread is affected by the overall ability weighting (Figure 47 below) shows that the common weighting type produces a ~1.5% increase for singing and dancing roles when used with the role learning engine, and ~1% when using the session learning engine.

Average		Singing Roles	Acting Roles	Dancing Roles	Any Roles
RL	Common	.463↑	.289↓	.408↑	.657↓
	None	.449↓	.292↑	.392↓	.673↑
	Vector	.449↓	.293↑	.392↓	.673↑
SL	Common	.458↑	.291↓	.410↑	.655↓
	None	.450↓	.292↑	.390↓	.673↑
	Vector	.450↓	.292↑	.390↓	.673↑
IW	Common	.458	.286	.396	.673↑
	None	.459	.285↓	.395	.674↑
	Vector	.457	.287	.399	.672

Figure 47 - Role spread by how overall ability is weighted in suitability calculations

However the opposite appears true for the spread of any roles, which shows ~1.7% decrease for the common weighting type. The difference for acting roles is barely significant, and the ideal weights engine showed only minor changes across the categories.

Average			Singing Roles	Acting Roles	Dancing Roles	Any Roles
RL	Subtract Costs From	Final	.454	.291	.396	.669
		Requirement	.453	.291	.397	.668
	Count Roles By	Geometric Mean	.454	.291	.396	.668
		Whole Roles	.454	.291	.397	.668
SL	Subtract Costs From	Final	.453	.292	.397	.667
		Requirement	.453	.292	.397	.667
	Count Roles By	Geometric Mean	.452	.292	.397	.667
		Whole Roles	.453	.292	.397	.667
IW	Subtract Costs From	Final	.458	.286	.397	.673
		Requirement	.458	.286	.397	.673
	Count Roles By	Geometric Mean	.458	.286	.397	.673
		Whole Roles	.458	.286	.397	.673

Figure 48 - Role spread by how costs are subtracted and roles are counted

Figure 48 above shows the role spread was unaffected by whether the existing role costs were subtracted from the final or requirement suitabilities, and whether roles were counted as whole or the geometric mean. This was true across all engine types.

For the session learning models, Figure 49 below shows that training once per session or with roles stockpiling improves the any role spread by ~0.5% over training every role individually, however the acting roles see a 0.2-0.4% decline. The singing roles category has no significant change.

Average Row Sample Size	Singing Roles	Acting Roles	Dancing Roles	Any Roles
EveryRoleIndividually	.450 27.000	.297 ▲ 27.000	.400 ▲ 27.000	.664 ▼ 27.000
EveryRoleStockpiling	.452 27.000	.288 ▼ 27.000	.392 27.000	.673 ▲ 27.000
OncePerSession	.451 9.000	.291 9.000	.393 9.000	.673 9.000

Figure 49 - Role spread by type of neural network training

Figure 50 below is a summary of the models which use the default or reasonable alternate parameters specified in section 4.2.4 Default engine parameters, except for the training schedule parameter as these are effectively different approaches. Due to a missing data point, one of the role learning models is unavailable, leaving 39 models to compare to the 2 heuristic baseline models and 3 human-picked results, with the full table available in Appendix L - Role spread results.

#	Model specification	Sing. Roles	Act. Roles	Danc. Roles	Any Roles
1	Human picked result	50.1%	34.5%	43.1%	73.1%
3	Heuristic baseline	47.9-48.0%	27.6%	39.7%	66.2%
5	Ideal weights	47.6-47.7%	27.5%	38.5-38.8%	67.2-67.3%
13	Session learning (role stockpiling)	46.3-46.4%	28.4-28.5%	38.1-38.2%	67.2%
21	Session learning (every role), subtract from final, whole roles	46.7%	27.8%	38.2%	67.3%
23	Role learning, count whole roles	46.7-46.9%	27.8%	38.1-38.2%	67.3%
27	Session learning (per session), subtract from final, whole roles	46.8%	27.7%	38.2%	67.3%
29	Session learning (every role), subtract from req'ment, whole roles	46.7%	27.8%	38.2%	67.2%
31	Role learning, subtract from requirement, count geometric roles	46.7%	27.7%	38.1%	67.3%
32	Session learning (every role), subtract from final, geometric roles	46.7%	27.8%	38.1%	67.3%
34	Session learning (per session), subtract from req'mnt, whole roles	46.8%	27.6%	38.2%	67.3%
36	Session learning (every role), subtract from req'ment, geom. roles	46.7%	27.7%	38.1%	67.2%
38	Role learning, subtract from final, count geometric roles	46.8%	27.7%	38.1%	67.3%
40	Session learning (per session), count geometric roles	46.7%	27.6%	38.1%	67.3%

Figure 50 - Summary of models by role spread

No models had a larger spread of roles than human results, regardless of category. The heuristic baseline was next best, at 2-6% behind the human result. The ideal weight models were next, on average 0.3% behind the heuristic for specific roles but 1% ahead for any roles. The neural models were tightly grouped, with dancing and any roles varying by only 0.1%, singing roles varying within 0.6% and acting within 0.9%.

4.4 Qualitative feedback

In addition to the numerical analysis, five of the production team members from CGS were provided a beta version of CARMEN for evaluation and user acceptance testing. They were given basic instructions of how to perform each step of the casting process, but otherwise left on their own to figure out how the software worked. In addition to instructions, example databases were provided representing the end of each step. This way if there were any issues completing a step, they could skip the remainder and start with a working database for the beginning of the next step.

Upon completing their evaluation, the production team members were surveyed. The full list of questions and responses are provided in Appendix M - UAT feedback survey and results, but an overview is provided below.

Figure 51 - Responses to "What are 3 things you liked about the software?"

Figure 51 above shows that the respondents liked CARMEN's ability to adapt to different configurations, the addition of tags and allowing roles to be in multiple items. Overall they described it as "quick" and 80% ranked it as significantly faster than the previous software.

Users highlighted the occasional small bugs in the user interface (see Figure 52 below), but understood that this is an early version and there is work yet to be done. A more notable complaint was a lack of information in the error messages describing applicants which don't have a role within a section. This is an area which will need improvement.

Figure 52 - Responses to "What are 3 things you disliked about the software?"

The users were also asked to score the importance and functionality of key features on a scale from 1 to 5, with the average scores shown in Figure 53 below.

Average	Important	Worked well
Allowing roles to be in multiple items	5.0 ↑	4.2
Highlighting of errors/inconsistencies	5.0 ↑	3.4
Balancing talent between casts	4.8	4.6
Easy to manually pick cast for roles	4.8	4.8
Easy data entry for Items and Roles	4.8	4.0
Easy data entry for Applicants and marks	4.8	4.0
Manual editing of selected cast	4.8	4.0
Storing/viewing Applicant photos	4.8 ↑	3.4
Keeping siblings in the same cast	4.6	4.6
Customisable Show Structure	4.4	4.4
Customisable Audition Criteria	4.4	4.6
Automatic balanced casting of multiple roles	4.2	4.4
Visibility of casting progress while casting roles	4.2	4.8
Automatically selecting cast based on past choices	4.0	4.4
Customisable cast Tags	3.8	4.4
Theming of software to represent your Show/Company	3.8	4.6
Automatic casting of a single role based on past choices	3.6	4.6
Complex Requirements	3.6	4.2

Figure 53 - Average scores for feature importance and implementation (out of 5)

While all listed features scored highly, five of the most important eight features were new to CARMEN, or significantly improved over the previous versions of the software. However two of these features, error highlighting and applicant photos, were found to work less well with an average score of 3.4 out of 5. These will require further improvement.

The usability of CARMEN was found to be quite easy, with the exception of the show configuration. There was a steep learning curve to understand what all the configuration options would mean, before you have even used the rest of the program. This is somewhat unavoidable, but thankfully this step is usually only done once for a production company, then the standard configuration re-used for subsequent shows. Overall, all respondents ranked CARMEN as at least slightly easier to use than the previous software.

Figure 54 - Responses to "Did the software give good casting recommendations?"

The casting recommendations were highly praised by all respondents, with most expressing surprise at how good they were (see Figure 54 above). The only times poor recommendations were found to occur was when the less important roles had been cast first, leaving all the talented applicants unavailable for an important role. Three respondents ranked CARMEN as much more intelligent than previous software, with the remaining two ranking it as slightly more intelligent. All respondents said they would also consider using CARMEN for productions other than CGS.

5. Discussion

This section will discuss the results found in section 4. Analysis and provide answers to the questions set out in section 3.1 Hypotheses.

5.1 First hypothesis

In order to test the first hypothesis, that using a SAT solving algorithm will produce more even balance between alternating casts than a heuristic algorithm, each of the SAT based selection engines were compared to the heuristic engine and human result by a number of the metrics.

The mean difference metric, analysed in section 4.1.1 Mean difference, was found to be significantly lower (which is better) for all SAT based approaches except the rank difference engine. The best approach found was chunked pairs, which achieved a mean difference of 1.2, 68% lower than the heuristic baseline at 3.8. The other engines achieved a mean difference between 1.4 and 2.5, universally lower than even the human produced result at 3.0.

The top 5 mean difference metric, analysed in section 4.1.2 Mean difference (Top 5), again found that all SAT engines except rank difference improved upon the heuristic and human produced results, at 5.2 and 3.4 respectively. The best approaches for this metric were top pairs, hybrid pairs and best pairs, with values between 1.6 and 1.7.

The rank difference metric, analysed in section 4.1.3 Rank difference, reaffirmed the mean difference finding that chunked pairs was the most effective, with the lowest rank difference of 4.7, an 80% reduction from the heuristic rank difference of 23.4, followed by hybrid pairs with a rank difference of 7.4. The remaining SAT based engines, except the rank difference engine, all performed well with rank differences between 10.7 and 12.4. This is an excellent result, as they were not specifically designed to minimise the rank difference metric, which indicates that they are genuinely solving the problem at hand.

In contrast, the rank difference engine, which was specifically designed to minimise the same rank difference cost function used in the metric, has performed poorly across all test cases by all metrics. My initial reaction to this was that there must be a simple bug in the code, but stepping through in a debugger showed no obvious failure, just that it was reaching the maximum time allotted before terminating. As the rank difference engine is not purely SAT, but also involves a branch and bound algorithm over a relatively slow to calculate cost function, it is important to limit the time as a full computation could take hours to complete. My next thought was that perhaps the maximum time was set too short, but experimentation with this value only found minor improvements in some cases, not a significant change overall. It is possible that the computation time for a reasonable result is too high, that the approach is fundamentally flawed in another way, or that there is still an unknown bug in my implementation, however further work is required to discern which of these is the root cause of the failure. This may be an interesting topic for future research.

The top 5 rank difference metric, analysed in section 4.1.4 Rank difference (Top 5), mimicked the result of the other top 5 metric with best pairs, top pairs and hybrid pairs all measuring between 1.8 and 1.9, compared to the heuristic baseline of 6.2. The chunked pairs approach also scored well with 3.0, 19% lower than the human produced result of 3.7. The rank difference engine was again the only approach worse than the heuristic, but in this instance “three’s a crowd” was slightly worse than the human produced result, at 4.0.

Comparing the results of the top 5 metrics to the whole list metrics, the heuristic is comparable to the human produced result only when analysing the whole list. The human produced results for top 5 metrics are instead closer to the SAT based engine results, confirming the anecdotal evidence that the production team focuses only on balancing the top of the talent pool.

Ranking the engines by show year found that the chunked pairs and hybrid pairs represented the best mean difference for 25 of the 29 test years, and the best rank difference for 28 of them. Looking at the top 5 metrics, best pairs achieved the best results most often, but in practical terms top pairs and hybrid pairs were equally as effective. This leads to the conclusion that for an overall balance between casts, the hybrid pairs approach is the best choice and should therefore be set as the default selection engine in CARMEN.

As a side metric, the time taken was analysed in section 4.1.5 Time taken to evaluate the relative speeds of the approaches. Of the highly ranked engines listed above, top pairs completed the fastest, in only 0.7 seconds, followed by chunked pairs in 3.8. Best pairs and hybrid pairs were an order of magnitude slower at 15.5 and 23.5 seconds respectively. While this doesn’t override the success of hybrid pairs as the best overall, and therefore default, selection engine, this does lead to two further conclusions.

1. If you know for sure that you only care about balancing the top talent or the whole list, then the best pairs or chunked pairs engines, respectively, should be used, as they will achieve a comparable result in a shorter time.
2. If you need the fastest operation, or have an unusually large data set, then the top pairs engine should be used, as it will complete the fastest while still performing significantly better than the heuristic engine or previous human results.

To formally address hypothesis #1, it has now been proven that using a SAT solving algorithm will produce a more even balance between alternating casts than a heuristic algorithm. Although it need only have been proven for one such algorithm, this has been found to be true for all SAT based engines tested except for rank difference, which, as discussed, has a yet to be identified fault.

5.2 Second hypothesis

In order to test the second hypothesis, that using a neural network will produce a more accurate recommendation of applicants for a role, than a heuristic algorithm, each of the neural network based

allocation engines were compared to the heuristic engine by a number of the metrics. Much of the analysis in section 4.2 Comparison of recommendation accuracy is dedicated to determining the correct default parameters for the implemented neural engines, the results of which are summarised in section 4.2.4 Default engine parameters. The discussion below will consider only models trained with these chosen parameters.

The comparison accuracy metric, analysed in section 4.2.2 Comparison accuracy, found that neural networks which train after each role is cast performed the best with an accuracy of 87.2%, which is 0.5% higher than the heuristic baseline at 86.7%. The neural networks which train once per session also performed well, at just below 87.2%, followed by those training by role with stockpiling at 87.0%. All neural network engines, except complex networks, performed better than even the ideal weight models, which were themselves better than the baseline heuristic at between 86.8% and 87.0% accuracy. This leads to the conclusion that the best neural network approach is the role learning engine, which should therefore be set as the default allocation engine in CARMEN.

Another observation worth examining is that at ~87%, the implemented models are substantial improvements on the 83% average in the pairwise POC examined in section 4.2.1 Pointwise vs pairwise. The main difference between the ML.NET package used for the POC and CARMEN implemented approaches is network complexity. Prior knowledge of the heuristic guided the role learning and session learning models to be implemented as single layer perceptrons, effectively performing weighted averages, whereas ML.NET would have operated more like the complex network approach, trying varying layers, structures and more advanced training algorithms in order to learn the casting operation. The fact that the simpler approaches beat the POC by such a large margin suggests that overfitting may have occurred, reducing the accuracy when given unseen test data, and put simply, that a more complex model is not required. This is consistent with the analysis of the structural parameters of the complex network approach, which found that less complex networks performed better, but does not fully explain why the complex network models performed poorly overall. Although future work could investigate exactly what the problem was, and likely use a fully featured ML library to make the complex networks a viable approach, there is currently no evidence that a more complex model will provide any benefit. This justifies the decision to implement the role and session learning approaches as single layer perceptrons.

The casting accuracy metrics, analysed in section 4.2.3 Auto-casting accuracy, were found to be less reliable than originally hoped, due to the propagation of errors, as one incorrect casting choice affects the applicants available for the next role. These metrics were also greatly affected by the order in which roles were cast, which showed a greater variation than any engine parameter. Unfortunately this is unavoidable as the human process of casting a show is inevitably unstructured, often involving going back and changing the initially cast applicants in retrospect, as well as being affected heavily by which role the team “feels” should be cast next at the time.

Caveats aside, the casting accuracy metrics did show that the ideal weight models were the most accurate, improving by ~0.3% over the best heuristic, in every category except similar relevant marks. This was followed by the neural network models which train after every role, which improved by ~0.2% over the best heuristic, also in every category except similar relevant marks, however it's worth noting that this is only true for models which counted roles as a geometric mean. The remaining models were within a range of ~0.5% for each category, showing no strong correlation but roughly matching the rank order of the comparison accuracy metric, with training once per session coming first, then training by role with stockpiling.

One of the likely problems with the casting accuracy metrics is that it effectively only compares the new role allocations to what were previously allocated manually, and does not make any judgement call on whether or not the new casting is better or worse. In retrospect, a better method to assess the quality of role allocations produced by CARMEN autonomously might have been to critically analyse the new casting of a show compared to the old casting and identify what casting choices were better, worse, or equivalent. This would have been impractical on the scale of the 50,992 combinations of engine types and parameters, but now that sensible defaults have been set, this may be plausible as an area for future research and analysis.

An interesting observation from the casting accuracy metric, which is not present in comparison accuracy, is that every neural model counting roles by a geometric mean shows a clear improvement over the equivalent model counting whole roles. This suggests that counting roles by a geometric mean more closely resembles the internalised mental process used by the production team in the human result. The opposite is true for the baseline heuristic models, which is consistent with the fact that the heuristic was originally designed when using software which only counted whole roles. This leads to the conclusion that geometric mean is the best method for counting roles and should be set as the default in CARMEN.

Given that the neural networks which learn after each role have been found to improve both the comparison accuracy and casting accuracy metrics, hypothesis #2 has now been formally proven, that is, that using a neural network will produce a more accurate recommendation of applicants for a role, than a heuristic algorithm. In addition to this proof for one such algorithm, it has also been proven, using the more reliable comparison accuracy metric, for all neural network based engines implemented except complex networks, which were found to not be a viable or beneficial approach.

It should be noted that although the measured improvement is only slight, it is statistically significant and shows that casting role allocation can be learned by a neural network with no prior knowledge of a heuristic which was developed from the expert knowledge gained by decades of personal experience.

5.3 Third hypothesis

In order to test the third hypothesis, that using a neural network will result in a more even spread of parts between applicants than a heuristic algorithm, each of the neural network based allocation engines were

compared to the heuristic engine by the role spread metric analysed in section 4.3 Comparison of role distribution.

Unlike the previous analysis of neural models by engine parameters, the role spread metric often showed insignificant or no change across the parameter options. Where it did, such as for the overall weight type, the results were conflicting across the categories. One such example is that the spread of singing and dancing roles were increased by common overall weights, whereas acting and any roles were decreased. This is in contrast to the accuracy metrics in section 4.2 Comparison of recommendation accuracy which showed a consistent improvement for vector or no overall weights. A similar conflicting result was seen for the training schedule parameter. The lack of correlation and conflicting results have led me to question whether the role spread was being affected by anything other than random change across the models. This is certainly an area which could use additional research, perhaps by examining role spread against model accuracy. I hypothesise that there may be a negative correlation between them, such that the less accurate the model is, the further the roles get spread.

Comparing the models trained with the chosen parameters defined in section 4.2.4 Default engine parameters, the neural models are tightly clumped within 0.2% for singing and acting roles, and 0.1% for dancing and any roles, with the only outlier being session learning with role stockpiling, which saw a 0.6% increase in acting roles spread over the others. While this could be considered a result, with session learning with role stockpiling improving over the heuristic baseline for acting and any roles, it achieves a significantly smaller spread in singing and dancing roles and is ranked 13th overall after all of the heuristic style approaches.

The highest values in all categories are achieved by the human-picked historical casting, which were on average 5% higher than the best neural models, and the heuristic baseline.

To formally address hypothesis #3, it has now been disproven that using a neural network will result in a more even spread of roles than a heuristic algorithm, however the juxtaposition of the human produced result against all measured automatic casting models provides some insight as to why.

The significantly higher role spread in the human-picked result suggests that it may not actually be possible to replace human choices in this part of the casting process, by a heuristic or otherwise. Allocating a role to a person of lesser talent, in order to increase the role spread, is an important judgement call by the production team. It is an assessment of whether an applicant is good enough that they can be taught, over the length of the rehearsal season, to perform the role adequately and should therefore be given a chance, or whether they simply don't have the ability required for a role and it should instead be allocated to someone of a higher talent who already has a role. There are many factors involved in a decision like this, something which a numerical casting program cannot hope to achieve, nor should it. At the end of the day, it is the production team's skills and hard work which will have to train this applicant to perform the role, and

whether or not they have the teaching skills to achieve this, or believe that they do, is just as relevant as the performance abilities of the applicant themselves.

Although the neural network models have failed to achieve the goal of the third hypothesis on their own, what CARMEN can do is provide a system to help the users make these human-centric choices. By providing a tool which helps the user understand their options, and save time by automating as much as possible of the casting process, their effort can instead be focussed on the small but specific choices like this, which only a human can do. The following is a list of ways in which CARMEN achieves this goal;

1. Configurable audition criteria, to mark applicants numerically, categorically or by ticking a box
2. A complex requirement system, with thresholds and combinational logic
3. Well-balanced alternative casts, to keep talent available where it is needed
4. High quality recommendations for role allocations, putting the options clearly in front of the user
5. Automated balanced casting of multiple roles at once, for less important roles or group roles
6. A clear, fast, and efficient GUI to reduce the effort and improve the speed of the casting process

In this way, I believe that CARMEN achieves the spirit of the goal of hypothesis #3.

6. Conclusion

This project set out to research and evaluate the use of SAT solving algorithms and machine learning neural networks in the domain of theatrical role allocation, specifically using SAT solving to balance talent between alternative casts and machine learning for intelligent recommendations. These have been identified as two of the key challenges in the automation of cast selection and role allocation.

With the goal of creating a fully functional casting program, CARMEN, short for Casting And Role Management Equality Network, has been created with customisable show structure, audition criteria and other parameters. This deliverable has been provided to Cumberland Gang Show, whose historical casting data from 1993 to 2021 was used as a case study for this research. It has also been released open source under the GPLv3 license and can be found at https://github.com/davidlang42/CARMEN.

Six SAT based approaches for balancing alternating casts have been implemented, tested and evaluated, with 5 of these being found to improve the balance of talent between alternative casts. This proves the first hypothesis of the research, that using a SAT solving algorithm will produce a more even balance between alternating casts than a heuristic algorithm. Of these, the “hybrid pairs” approach was found to be the most effective overall and has been incorporated into CARMEN as the default selection engine.

Three neural network approaches for intelligent role recommendations have been implemented, tested and evaluated, with 2 of these being found to improve the recommendation’s accuracy at predicting historical casting results in the case study. This proves the second hypothesis, that using a neural network will produce a more accurate recommendation of applicants for a role, than a heuristic algorithm. Of these, the “role learning” approach was the most accurate and has been incorporated into CARMEN as the default allocation engine.

The same neural network approaches have been tested and evaluated for their ability to spread roles between applicants. None were able to improve on the spread of roles generated by the heuristic algorithm reproduced from the previous Casting Expert software, and not even heuristic style algorithms were able to match the role spread of the human picked historical casting. This disproves the third hypothesis, that using a neural network will result in a more even spread of roles than a heuristic algorithm, but anecdotal evidence puts doubt as to whether a program could, or even should, attempt to replicate all human decisions in the casting process.

Instead, it is suggested that creating a logical and effective casting automation tool, such as CARMEN, will reduce the total work required to cast a show, leaving the user with more time and energy to focus on these crucial decisions which are best left to humans. This was reinforced by the feedback of five members of the CGS production team who tested a beta version of CARMEN, which they described as “quick”, “easy” and “flexible”, universally ranking it as faster, easier to use and more intelligent than the previous software.

I believe this research, and the deliverable software package CARMEN, has improved the quality of, and greatly reduced the time required to complete, the casting process of medium-large theatrical productions. It will first be used for a real world production by Cumberland Gang Show in December 2021, as they start auditions for their July 2022 performance, and will help the volunteer production team to share roles fairly amongst applicants, leaving them to focus on what they do best; developing the talent, creativity and passion for the arts of the youth (and youth-at-heart) of Sydney, Australia.

References

Alway, A., Zamri, N. E., Karim, S. A., Mansor, M. A., Mohd Kasihmuddin, M. S., & Mohammed Bazuhair, M. (2021). Major 2 Satisfiability Logic in Discrete Hopfield Neural Network. *International Journal of Computer Mathematics*, *ahead-of-print*(ahead-of-print), 1–25. https://doi.org/10.1080/00207160.2021.1939870

Budinich, M. (2019). The Boolean SATisfiability Problem in Clifford algebra. *Theoretical Computer Science*, *784*, 1–10. https://doi.org/10.1016/j.tcs.2019.03.027

Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, *5*(7), 394–397. https://doi.org/10.1145/368273.368557

Davis, M., & Putnam, H. (1960). A Computing Procedure for Quantification Theory. *Journal of the ACM*, *7*(3), 201–215. https://doi.org/10.1145/321033.321034

Displayr (2021). Displayr (build 1.0.28345) [Cloud-based computer software]. Retrieved from https://www.displayr.com

Ganzinger H., Hagen G., Nieuwenhuis R., Oliveras A., Tinelli C. (2004) DPLL(*T*): Fast Decision Procedures. In: Alur R., Peled D.A. (eds) Computer Aided Verification. CAV 2004. Lecture Notes in Computer Science, vol 3114. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-27813-9_14

Goswami, A. (2018). *Machine-Learned Ranking Algorithms for E-commerce Search and Recommendation Applications*. ProQuest Dissertations Publishing. https://search.lib.uts.edu.au/permalink/61UTS_INST/1ibc883/cdi_proquest_journals_2091525981

Guo, W.-S., Yang, G.-W., Hung, W. N. N., Song, X. (2013). Complete Boolean Satisfiability Solving Algorithms Based on Local Search. *Journal of Computer Science and Technology*, *28*(2), 247–254. https://doi.org/10.1007/s11390-013-1326-4

Larrosa, J., Nieuwenhuis, R., Oliveras, A., & Rodríguez-Carbonell, E. (2009). Branch and Bound for Boolean Optimization and the Generation of Optimality Certificates. In *Theory and Applications of Satisfiability Testing - SAT 2009* (Vol. 5584, pp. 453–466). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-02777-2_42

Liu, M., Zhang, F., Huang, P., Niu, S., Ma, F., & Zhang, J. (2020). Learning the Satisfiability of Pseudo-Boolean Problem with Graph Neural Networks. In *Principles and Practice of Constraint Programming* (Vol. 12333, pp. 885–898). Springer International Publishing. https://doi.org/10.1007/978-3-030-58475-7_51

Lorrentz, P. (2015). *Artificial neural systems : principle and practice* . Bentham Science Publishers Limited. https://doi.org/10.2174/97816810809011150101

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: engineering an efficient SAT solver. *Proceedings of the 38th Annual Design Automation Conference*, 530–535. https://doi.org/10.1145/378239.379017

Nikolic, M., Maric, F., & Janicic, P. (2013). Simple algorithm portfolio for SAT. *Artificial Intelligence Review*, *40*(4), 457–465. https://doi.org/10.1007/s10462-011-9290-2

Ozolins, E., Freivalds, K., Draguns, A., Gaile, E., Zakovskis, R., & Kozlovics, S. (2021). *Goal-Aware Neural SAT Solver*. https://arxiv.org/abs/2106.07162

Pedrycz, W., Succi, G., & Shai, O. (2002). Genetic-fuzzy approach to the Boolean satisfiability problem. *IEEE Transactions on Evolutionary Computation*, *6*(5), 519–525. https://doi.org/10.1109/TEVC.2002.804915

Selsam, D. (2019). *Neural Networks and the Satisfiability Problem*. ProQuest Dissertations Publishing. https://stacks.stanford.edu/file/druid:jt562cf4590/dselsam_dissertation_final-augmented.pdf

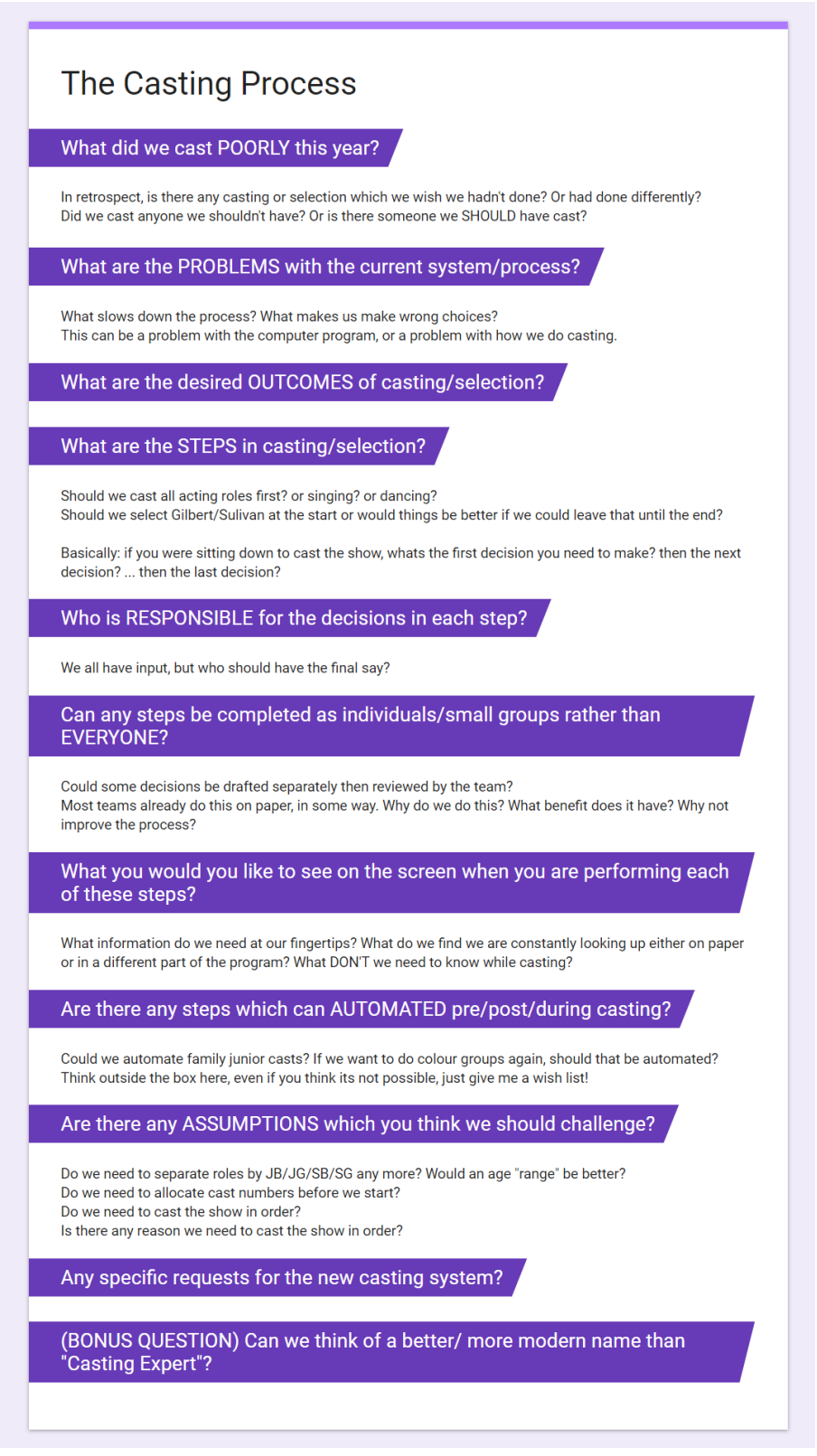
Wang, B., & Klabjan, D. (2017). *An Attention-Based Deep Net for Learning to Rank*. https://arxiv.org/abs/1702.06106

Zaikin, O. (2017). A parallel SAT solving algorithm based on improved handling of conflict clauses. *Procedia Computer Science*, *119*, 103–111. https://doi.org/10.1016/j.procs.2017.11.166

Appendices

Appendix A - Previous user feedback

Over the past 10 years, various formal and informal feedback has been submitted by the CGS production team about the existing Casting Expert for Windows software package. An example of the formal feedback is shown in Figure 55 below, followed by a list of dot points summarising their complaints and requests grouped by approximate timeframe.



The Casting Process

What did we cast POORLY this year?

In retrospect, is there any casting or selection which we wish we hadn't done? Or had done differently?
Did we cast anyone we shouldn't have? Or is there someone we SHOULD have cast?

What are the PROBLEMS with the current system/process?

What slows down the process? What makes us make wrong choices?
This can be a problem with the computer program, or a problem with how we do casting.

What are the desired OUTCOMES of casting/selection?

What are the STEPS in casting/selection?

Should we cast all acting roles first? or singing? or dancing?
Should we select Gilbert/Sulivan at the start or would things be better if we could leave that until the end?

Basically: if you were sitting down to cast the show, whats the first decision you need to make? then the next decision? ... then the last decision?

Who is RESPONSIBLE for the decisions in each step?

We all have input, but who should have the final say?

Can any steps be completed as individuals/small groups rather than EVERYONE?

Could some decisions be drafted separately then reviewed by the team?
Most teams already do this on paper, in some way. Why do we do this? What benefit does it have? Why not improve the process?

What you would you like to see on the screen when you are performing each of these steps?

What information do we need at our fingertips? What do we find we are constantly looking up either on paper or in a different part of the program? What DON'T we need to know while casting?

Are there any steps which can AUTOMATED pre/post/during casting?

Could we automate family junior casts? If we want to do colour groups again, should that be automated?
Think outside the box here, even if you think its not possible, just give me a wish list!

Are there any ASSUMPTIONS which you think we should challenge?

Do we need to separate roles by JB/JG/SB/SG any more? Would an age "range" be better?
Do we need to allocate cast numbers before we start?
Do we need to cast the show in order?
Is there any reason we need to cast the show in order?

Any specific requests for the new casting system?

(BONUS QUESTION) Can we think of a better/ more modern name than "Casting Expert"?

Figure 55 - Questions from the Google Forms survey in 2018

2013-2015 (during casting process)

- “Select Cast” UI is slow to respond on load/sort/save
- Import previous year’s audition marks
- Default values for Scout/Guide, New/Ex-Cast, Voice Type should be NULL, rather than one of the values
- Multiple users using the same database would be advantageous sometimes
- Importing applicants is clunky/unreliable and opaque
    - Don’t hard-code field/column names/format
    - More transparent about what is imported successfully/ what went wrong
    - Import audition group, then use this to sort the prefilled applicants list
- Better support for double-casting a junior
    - Automatic creation (rather than having to manually make a copy of them)
    - Ticking/unticking in selection window, make sure both are in sync
- Add default sorting to all queries (by cast number and/or name)
- Right-click menu while selecting cast: Select All, Unselect All, Auto-select cast
- Allow one character to be in many items

2018 (survey as shown above)

- Some gender neutral roles, where applicable
- Sing/Act/Dance teams enter pre-casting suggestions before casting day
- Know who has the mics while we are casting
- Make sure each general cast group has some mics on stage
- Colour groups are good on the floor but need to be allocated during casting
- Talent buddies for juniors works very well, but this needs to be incorporated into the casting system
- Balance talent between general cast groups
- Balance talent between g/s junior casts
- As well as photo & age, list someone’s other lead roles while casting rather than just a count
- Better reporting & graphs
- Cast key roles first, then filter down to general roles, no specific need to cast the show in order
- Don’t show the full list of cast on each screen, filter out those who are ineligible (already cast, etc)
- Age range criteria for some roles
- Variety in lead roles across talent pool (not just the best person each time)
    - Still needs to be a good show, but cast development is important
    - Give people a chance but without putting them into a role they won’t be capable of
- Production team members need to be cast in items which they are producing
- Cast “Character Leaders” (one or a few) for each general cast group- this would be someone which the production team can use to assist coordinating a group of people who are supposed to act as a certain character or do a certain routine
- Adjustment done by program (production mark to be entered raw)

- More info on person from auditions, possibly voice type as multi-select
- Allow criteria based on many things including voice type

2019 (during casting process)

- Ability to change cast numbers (needs to update buddycastID)
- Include date and time on all reports
- Calculate age in July from DOB rather than manual entry
- Choose dance team for tied marks from the same order as the final report prints
- Family names CANNOT be buddy ever
- Family names in SAME CAST

2020 (during casting process)

- Lock pairs (for microphones) based on conditions, eg. singing, acting roles or lock actual number buddies for non-roles
- Show mics in each item while casting
- Add layer to casting completeness tree, showing roles within items
- Grey-out ineligible cast in the “Select Cast” UI, and high-light others based on cast groups

2021 (discussion of new software for this project)

- Cast replacements on drop out
    - Replace with someone not selected
    - Satisfy critical roles of dropped out to recast with minimal changes
    - Ability to save version, and generate casting change reports since change
    - See what people are current cast in/as if BRAC or SAME (ie. not have to wait until you cast them and it gives you the report)
- Roles in multiple items
    - Its okay to not have “characters numbers”
    - Always list roles alphabetically by name
- Custom reporting (including csv)
    - Eg. for wardrobe- list of (item, person, character)
- Model mic list as cast group
    - Maximum count of mics
    - Generate mic list of anyone with a Singing Mark > 70
    - Hard requirement for singing roles
- Import from previous shows
    - Single applicants including photos & abilities (where criteria match)
    - Requirements, section types, cast groups
- Auto casting rules
    - Juniors different ages

    - Eg. Jun Girl < 14
    - Eg. Jun Boy < 15
- Two junior casts as an optional feature
    - So it can be enabled/disabled year by year rather than being hard coded
    - Ability to treat a double cast junior as 1 entity
    - Ensure talent buddies get cast identically (to help mic list)
    - Options to buddy based on height, talent, or match talent within height ranges
- Allow choice over how the cast numbers are allocated, not always height
- Desktop app is preferable to cloud-based because
    - casting app is generally used heavily for a week, once a year, not regularly
    - cloud requires maintenance effort/on-going costs
    - need to be able to run it offline without a reliable connection
    - personal data privacy could be a concern, easily avoided
    - general user preference for desktop app
- More analytics
    - number of special parts per person (freq. dist. & cum. prob.)
    - number of parts against mark (scatter) for each special category
    - total number of special parts per category (absolute and per cast member)
    - cast list of name, mark, number of parts sorted by mark for each special category

Appendix B - Functional requirements

A full list of functional requirements of the CARMEN software package are included below, grouped into logical sections. Where a function existed in a previous CE version but has been excluded from CARMEN, a reason is referenced, otherwise the CARMEN column indicates if this feature is part of the MVP, or planned for a future release. These specifications were written in conjunction with Dr Robert Lang, the producer/director of Cumberland Gang Show.

Overview

Function	Details	CE4.1	CE5.3	CARMEN
Reopen recent show	On start, reload the last show which was open on this computer (via registry)	✘	✓	✘*1
Startup prompt	On start, provide an option to start a new show, or open an existing show, including a recent show's list of at least 3	✘	✘	MVP
Default to next year's show	On create new, default to the next calendar year (if currently July or later)	✘	✓	✘*2
Applicant summary	On the main page, show a summary of applicants and cast selected, grouped by Cast Group, with totals	✓	✓	MVP
Process progress	The main screen should also show which steps have been completed, and which step you are up to. eg. Applicant import, Applicant marks, Cast Selection, Role Allocation, etc.	✘	✘	MVP
Theming	Prominently include the show's name and logo on the main screen	✘	✓	MVP
Multi-user	Allow multiple instances/ users to be connected to the same database file, both performing operations reasonably concurrently. At worst, it should tell you there was a conflict from another user, or silently overwrite changes, but it should definitely not crash or cause inconsistent/corrupted data. This will specifically allow the following use cases: - Many users viewing the data (sorted in their own way) while allocation is performed by one - Specialist cast groups (eg. Dance Team, Singing Team, Microphones) to be specified by the individuals responsible, prior to the day of casting	✘	✘	FUTURE
Desktop application	Desktop app is preferable to cloud-based because - casting app is generally used once a year, not regularly - cloud requires maintenance effort and on-going costs - need to be able to run it without a reliable internet connection - best to avoid personal information in the cloud for privacy reasons	✓	✓	MVP

	- general user preference for desktop app			
Import show metadata	Now that much of the business logic is stored in settings and structures (eg. Cast Groups, Criteria, Requirements), this will need to be easily transferred from one show to another, to avoid having to set up everything from scratch when the rules about your show structure has not changed.	✘	✘	FUTURE
Priority-based casting process	Cast key roles first, then filter down to general roles, no specific need to cast the show in order. The order should be suggested/lead by the casting system to choose the most contentious/pivotal or limited option roles first.	✘	✘	MVP

***1** Excluded due to proper file browsing, rather than a fixed folder path for database storage*

***2** Excluded due to being GS-specific business logic*

Cast groups

Function	Details	CE4.1	CE5.3	CARMEN
Add/edit Cast Groups	Ability to add/edit cast groups, including names, requirements, icon	✘	✘	MVP
Change requirements	Previously the rule about what age a Junior became a Senior was hard coded, it would be much better if this were an option based on age and gender	✘	✘	MVP
Non-primary cast groups	Previously cast groups were a single set, of which you could only be allocated into one. This functional is maintained by Cast Groups set as "Primary", making them mutually exclusive, however it would be helpful to also have non-primary cast groups which anyone can be allocated into, such as "Dance Team", "Production team", or "Microphone List"	✘	✘	MVP

Applicant data entry

Function	Details	CE4.1	CE5.3	CARMEN
Import applicant details (static)	Import applicant data to prefill (not yet auditioned)	✘	✓	FUTURE
Import applicant details (dynamic)	As above, but without hard coding column/field names. It should provide a reasonable UI to import data and visual indications of progress and errors encountered. This should allow importing all applicant options, including a predefined mark (eg. production adjustment), external identifier (eg. Hub Id) or cast group (eg. Production Team member).	✘	✘	FUTURE
Add applicant from prefill	Add new applicant, from prefill data	✘	✓	MVP

Sort prefill data	When adding new applicants from prefill, it should be easy to sort by audition group	✘	✘	MVP
Add new applicant	Add new applicant (no prefill)	✓	✓	MVP
Import applicant marks	Import applicant details, photos and marks from a previous show, where the assessed abilities are the same	✘	✘	FUTURE
View/edit applicant details	First Name Last Name Gender Photo Date of Birth Age at show date (calculated, read only) Audition Group	✓	✓	MVP
View/edit applicant marks	Voice Type Height Singing Mark Acting Mark Dancing Mark Production Mark Overall Mark (calculated, read only)	✓	✓	MVP
View/edit applicant demographics	Scout/Guide flag New/Ex-Cast flag	✓	✓	✘*2
Null default values	Avoid having misleading default values selected, eg: Gender Scout/Guide New/Ex-Cast Voice Type	✘	✘	MVP
View/edit applicant outcomes	Accepted flag (read only) Cast Number, including G/S (read only, toggle G/S) Primary Cast Group Dance Team flag	✓	✓	MVP
Applicant notes	View/edit multi-line free text applicant notes	✘	✘	MVP
Traverse applicants	Step between applicants (first, prev, next, last) by name alphabetically or by audition group	✓	✓	✘*3
Search applicant list	Select applicant from a list (showing LAST, FIRST name), filtered by search box (string contains)	✘	✓	MVP
Delete applicant, keeping prefill	Keep prefill data available to re-add later	✘	✓	✘*4
Delete applicant	Remove applicant and prefill	✓	✓	MVP

***3** Excluded due to being an outdated UI structure*

***4** Excluded due to removal of differentiation between applicants and auditioned cast*

Cast selection

Function	Details	CE4.1	CE5.3	CARMEN
Specify total cast required	Specify total cast required by Cast Group (taking into account two junior casts)	✓	✓	MVP
Specify dance team required	Specify dance team required by Cast Group	✓	✓	MVP
Auto-select the cast into Cast Groups	Select the cast, setting - accepted flag based on overall mark - dance team flag based on dance mark	✓	✓	MVP
Allocate cast numbers by height	- positive cast numbers based on height (when accepted) - negative cast number based on overall (when not accepted)	✘	✓	MVP *5
Allocate cast numbers by other metric	For example, age, talent	✘	✘	MVP *6
Warn if siblings in different casts	Warning report after selecting if same surname applicants have different cast GS	✘	✓	✘*2
Lock certain applicants together	Set a requirement (not sure where) that person A and person B are in the same cast group (ie. keep siblings in the same cast where possible, and never buddies)	✘	✘	MVP
Optional two-junior casts	It should be possible to setup/enable/disable the feature of having 2 junior casts, which are buddied	✘	✘	MVP
Height buddies	Junior cast buddies allocated based on height	✘	✓	MVP
Talent buddies	Junior cast buddies allocated based on talent	✘	✘	MVP
Complex buddies	Junior cast buddies allocated based on more complicated rules like matching talent within height ranges	✘	✘	FUTURE
Double cast juniors	Ability to treat a double cast junior as 1 entity, ie. have a junior which is in both casts of a two-junior cast system, and only have to add them once and cast them once	✘	✘	FUTURE
Balance talent between casts	This should attempt an even distribution of marks of *all* abilities between alternating casts	✘	✘	MVP

***5** Excluding negative cast numbers, which have been removed as a concept*

***6** Excluding order by age, which may be added in a future release*

Applicant reporting

Function	Details	CE4.1	CE5.3	CARMEN

Generate standard reports	One button to generate standard applicant reports - Alphabetical- (Cast Number, Cast GS, Name) sort (Last Name, First Name) - Cast Number- (Cast Number, Cast GS, Name) sort (Cast Number, Cast GS) - Singing- (Cast Number, Cast GS, Name, Singing Mark, Voice Type) sort (Singing Mark), group (Cast Group) - Acting- (Cast Number, Cast GS, Name, Acting Mark) sort (Acting Mark), group (Cast Group) - Dancing- (Cast Number, Cast GS, Name, Dance Mark, Dance Team) sort (Dance Mark), group (Cast Group) - Overall- (Cast Number, Cast GS, Overall Mark) sort (Overall Mark), group (Cast Group) - Reserve Name- (Cast Number, Name) sort (Last Name, First Name) - Reserve Overall- (Cast Number, Name, Overall, Phone Number) sort (Overall Mark), group (Cast Group) - Summary- (Cast Number, Cast GS, Name, Age, Singing Mark, Acting Mark, Dance Mark, Production Mark, Overall Mark) sort (Last Name, First Name)	✓	✓	FUTURE
Custom/individual reports	Custom reports of (All, Accepted, Rejected) Applicants sorted by Name, Singing, Acting, Dancing, Production, Overall, Cast Number, *Summary*) optionally grouped (Cast Group)	✓	✓	FUTURE
Formatted reports	RTF, 2 columns (except summary), header containing show name and date generated	✘	✓	FUTURE
Templated reports	Some way to use external templates to format reports, so that more reporting and custom formats can be created without code changes	✘	✓	FUTURE
Applicant statistics	Male/Female ratio count, % female Ex/new cast ratio count, % new cast Scout guide ratio count, % guide Senior/junior ratio count, % junior % of applicants to got in % of girls who applied got in % of boys who applied got in % of guides who applied got in % of scouts who applied got in % of ex-cast who applied got in % of new-cast who applied got in	✓	✓	✘*2
Export to CSV	Export to CSV (GangID, ServiceCastId, LastName, FirstName, Accepted flag, Cast Group, Cast Number, Cast GS)	✘	✓	FUTURE

Report theming	Prominently include the show's name and logo on generated reports	✘	✘	FUTURE

Show structure

Function	**Details**	**CE4.1**	**CE5.3**	**CARMEN**
Add/edit/delete item details	Bracket Item number Item name Required count for each Cast Group Total required in Item	✓	✓	MVP
Change item order	Change item order (previously didn't update bracket based on new item position, but the new version will because it is a proper tree structure)	✘	✓	MVP
Highlight incorrect totals	Highlight item totals that aren't the sum of the counts by cast group Highlight role totals that aren't the sum of the counts by cast group	✘	✓	MVP
Auto-fix totals	Double click wrong item total to update with correct sum Double click wrong role total to update with correct sum	✘	✓	✘*3
Highlight incorrect sums	Highlight item count by groups that aren't the sum of the item's roles' count by group	✘	✓	MVP
Bracket sums	Show count by group sums for the selected item's bracket	✘	✓	✘*3
Highlight incomplete brackets	Highlight bracket count by groups that aren't the total cast group numbers	✘	✓	MVP
Add/edit/delete role details	Character letter Character name Required count for each Cast Group Total required for role	✓	✓	MVP
Set role requirements	Singer required Actor required Dancer required Voice type required	✓	✓	MVP
Complex role requirements	Eg. Age, Height, Gender, logical combinations (and/or/xor/not), including ranges of ability and fixed values	✘	✘	MVP
Roles in multiple items	Allow roles to be in multiple items. Roles no longer need to be ordered within an item, instead they will always be listed in alphabetical order.	✘	✘	MVP

Gender neutral roles	It would be ideal if there was a way to avoid specifying gender for some roles, as not every role requires a specific gender. Unfortunately it is likely that gender is baked into a Cast Group specification, which then means the count needs to be set for that Cast Group, inherently setting the gender. If a workaround for this is not plausible, the required count by cast group for a role will just need to be changed on the fly to achieve this.	✘	✘	FUTURE
Nested sections	Proper tree structure rather than a single layer of items grouped by bracket. Each section is a certain section type, which has different rules.	✘	✘	MVP
Add/edit section types	Add/edit/delete section type details: name, icon, AllowNoRoles, AllowMultipleRoles. That way, each type of section can have its own rules about what it means in the structure.	✘	✘	MVP
Variable role counts	Have a flag on each role indicating whether this role has variable counts. If true, then the casting engine may programmatically change any non-zero counts to meet other requirements, however it will never change a count to zero.	✘	✘	FUTURE

Role allocation

Function	**Details**	**CE4.1**	**CE5.3**	**CARMEN**
Show Bracket progress	For the selected Bracket, show: count by group & total required, count by group & total allocated	✓	✓	MVP
Show Item progress	For the selected Item, show: count by group & total required, count by group & total allocated	✓	✓	MVP
Show Role progress	For the selected Role, show: count by group & total required, count by group & total allocated	✓	✓	MVP
Show Role requirements	For the selected Role, show required abilities (Singer, Actor, Dancer, Voice Type)	✓	✓	MVP
Edit Role requirements	Allow editing of required abilities, with a confirmation prompt	✘	✓	FUTURE
Progress summary	Show tree of completed casting with check boxes: Show/Bracket/Item/(not character, but it could have)	✘	✓	MVP
Disable item check	Checkbox to disable consecutive item check- used at act boundaries, between linked items (ie. the same role exists in 2 consecutive items) and finale (because no costume change)	✘	✓	✘*7
Time gap rather than consecutive items check	Store the length of time for each item, and allow the condition to be a minimum time between roles rather than simply next/prev item.	✘	✘	FUTURE

Select cast for a Role from a list	Picked, Suitability, Cast Num, Name, Group, Singing Mark, Singing Parts, Acting Mark, Acting Parts, Dancing Mark, Dancing Parts, Overall Mark, Brac flag (already cast in bracket), Same flag (already cast in item), Prev flag (cast in previous item), Next flag (cast in next item)	✓	✓	MVP
Context menu in list	Right-click for quick access to Select All, Deselect All, Clear Cast	✘	✘	FUTURE
Calculate suitability	Calculate suitability percentage based on required abilities regardless of availability flags (the flags are only used to avoid auto-picking)	✓	✓	MVP
Sort list by suitability	Sort by suitability (by default)	✓	✓	MVP
Sort list by other fields	Change sort to (Suitability, Singing Mark, Acting Mark, Dancing Mark, Production Mark, Overall Mark, Num of Singing Parts, Num of Acting Parts, Num of Dancing Parts, Cast Number, Picked)	✘	✓	FUTURE
Auto-pick cast	Auto-picks cast if no one is allocated yet, you just need to click accept to save it	✘	✓	MVP
Clear role allocation	Clear button to clear the current casting of this role, which enables auto-pick for next selection of cast	✘	✓	MVP
Count allocation while picking	Every time an applicant is picked or unpicked, show the current number of applicants picked in their Cast Group	✘	✓	MVP
View Photo while casting	Double click name to see photo (in a pop-up window)	✘	✓	FUTURE
Incomplete allocation warning	Warning report on save role allocation if: - not enough cast selected to meet requirement - anyone selected in same bracket - anyone selected in next/prev item - anyone selected in same item (and that they got taken out of their previous role)	✘	✓	✘*3
Responsive selection UI	A known existing problem with CE is a slow to respond UI when "Select cast" is chosen, as it is generating/populating and creating controls for the list every time, on the fly. It is similarly slow to save and clear.	✘	✘	MVP
Balanced talent between general cast roles	When there is only one general cast role left after allocating all the special roles, everyone who does not yet have a role in that bracket will be cast there. However, if there are more than 1 such general role, whichever you choose first will get the more talented people by default. This feature is to ensure that talent is spread evenly between general cast roles.	✘	✘	MVP

Hide unavailable applicants	Don’t show the full list of cast on each screen, filter out those who are ineligible (already cast, etc). Perhaps optionally, in case the user is trying to find someone specific and needs to understand why they can't be chosen. But if they are shown, make sure they are greyed or crossed out	✘	✘	MVP
List other lead roles while casting	In the "select cast" list, include age and list someone’s other lead roles rather than just a count. Note: This would probably make the list far too busy, so it probably means having easy access to this information- such as where the number of lead roles is, click and it shows a context menu listing them.	✘	✘	FUTURE
Show age in casting list	In the "select cast" list, include age as a column, or another easily accessible way	✘	✘	FUTURE
Show Cast Group Icons in casting list	In the "select cast" list, include icons representing the cast groups this cast member is part of (eg. Microphone, Dance team, Production team)	✘	✘	MVP
Spread lead roles around	Ensure, somehow, that lead roles are spread across those with a reasonable talent level, rather than always picking the single best person for every role.	✘	✘	MVP
Lock pairs	Some method to lock pairs (buddies) together for singing roles, such that the mic plot will be possible.	✘	✘	FUTURE
Sound reasoning	For all requirements, including abilities or availability flags, if an applicant is not suitable for a role, a mouseover/click/or reasonable action should provide a plain english description of why they aren't suitable. Eg. "David is already cast in this item, and his singing mark is too low."	✘	✘	MVP

***7** No longer required as roles can be in multiple items*

Verification

Function	**Details**	**CE4.1**	**CE5.3**	**CARMEN**
Every ROLE has the required CAST	List failures by item: Character X cast group Y was A should have been B	✓	✓	MVP
Every PERSON is in ONE item per BRACKET	List failures by bracket: Cast number, name (X items)	✓	✓	MVP
No PERSON is in TWO consecutive ITEMS	List failures by pairs of consecutive items: Cast number, name	✓	✓	MVP
Required ROLES for an ITEM add to the	List failures by item: Item Cast Group: X, Sum of Chars Cast Group: Y	✘	✓	MVP

CAST required for that ITEM				
All character & item TOTALS are the correct sum of the cast groups	List failures by item (total: X, count by groups A,B,C,D), then role (total: X, count by groups A,B,C,D)	✘	✓	MVP

Casting reports

Function	Details	CE4.1	CE5.3	CARMEN
Generate standard reports	One button to generate standard casting reports - Casting by Item- (character letter, total count, name, count by groups, [list of cast members allocated]) group (item number, name) - Casting by Person- (Bracket, Item number, name, role) group (cast number, cast GS, applicant name, buddy name) - List of characters- (total count, character letter, name, count by groups) group (item number, name)	✓	✓	FUTURE
Formatted casting slips	"Casting by Person" exports as RTF, with exactly 4 casting slips per page and no headers or footers	✘	✓	FUTURE
Formatted casting report	"Casting by Item" exports as RTF, with header containing show name, page number of total, and page breaks after every item	✘	✓	FUTURE
Advanced analytics	Such as - Number of special parts per person (freq. dist. & cum. prob.) - Number of parts against mark (scatter) for each special category - Total number of special parts per category (absolute and per cast member) - Cast list of name, mark, number of parts sorted by mark for each special category	✘	✘	FUTURE
Report theming	Prominently include the show's name and logo on generated reports	✘	✘	FUTURE
Export CSV	Data dump of casting details, in a reasonable format to import into something else. Specific example: for the wardrobe team- a fully enumerated list of (item, person, character) tuples	✘	✘	FUTURE

Casting changes

Function	Details	CE4.1	CE5.3	CARMEN
Swap cast	Choose an accepted cast member which has dropped out, and a rejected cast member who will take their place. The casting system	✘	✓	FUTURE

	will swap all cast roles, cast numbers including G/S, and accepted/rejected flags between these two applicants.			
Cast reallocation	Similar to the initial casting process, a dropped out cast member will need to have their lead roles reallocated. The system should guide a user through finding replacements, which attempts to upgrade existing leads into higher roles, rather than finding the best person currently without a lead role in the bracket. It should be the goal to satisfy the critical roles with minimal cast changes.	✘	✘	FUTURE
Versioning	Ability to save version, and generate casting change reports since change	✘	✘	FUTURE

Appendix C - ShowModel data descriptions

The CARMEN application is built around a data model representing a show called the ShowModel. The signatures (excluding code) for the C# data classes are included in this appendix grouped by namespace, then in alphabetical order.

Carmen.ShowModel

Image.cs

```
public class Image : INameOrdered, INamed, IComparable
{
    public Image();

    [Key]
    public int ImageId { get; }
    public string Name { get; set; }
    public byte[] ImageData { get; set; }
}
```

Carmen.ShowModel.Applicants

Ability.cs

```
public class Ability : INotifyPropertyChanged
{
    public Ability();

    public virtual Applicant Applicant { get; set; }
    public virtual Criteria Criteria { get; set; }
    public uint Mark { get; set; }

    public event PropertyChangedEventHandler? PropertyChanged;

    protected void OnPropertyChanged([CallerMemberName] string? name = null);
}
```

AlternativeCast.cs

```
public class AlternativeCast : INameOrdered, INamed, IComparable,
INotifyPropertyChanged
{
    public AlternativeCast();

    [Key]
    public int AlternativeCastId { get; }
    public string Name { get; set; }
    public char Initial { get; set; }
    public virtual ICollection<Applicant> Members { get; }

    public event PropertyChangedEventHandler? PropertyChanged;

    protected void OnPropertyChanged([CallerMemberName] string? name = null);
}
```

Applicant.cs

```
public class Applicant : INotifyPropertyChanged, IComparable<Applicant>
{
    public Applicant();

    public string ExternalData { get; set; }
    public int? CastNumber { get; set; }
    public virtual CastGroup? CastGroup { get; set; }
    public string Notes { get; set; }
    public virtual ICollection<Ability> Abilities { get; }
    public virtual Image? Photo { get; set; }
    public bool IsAccepted { get; }
    public DateTime? DateOfBirth { get; set; }
    public Gender? Gender { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public virtual ShowRoot ShowRoot { get; set; }
    [Key]
    public int ApplicantId { get; }
    public virtual SameCastSet? SameCastSet { get; set; }
    public virtual ICollection<Tag> Tags { get; }
    public virtual ICollection<Role> Roles { get; }
    public uint? Age { get; }
    public string Description { get; }
    public bool IsRegistered { get; }
    public string? CastNumberAndCast { get; }
    public virtual AlternativeCast? AlternativeCast { get; set; }

    public event PropertyChangedEventHandler? PropertyChanged;

    public static bool HasAuditioned(bool is_registered, IEnumerable<Ability>
applicant_abilities, IEnumerable<Criteria> all_criterias);
    public uint? AgeAt(DateTime date);
    public int CompareTo(Applicant? other);
    public string FormattedMarkFor(Criteria criteria);
    public bool HasAuditioned(IEnumerable<Criteria> all_criterias);
    public uint MarkFor(Criteria criteria);
    protected void OnPropertyChanged([CallerMemberName] string? name = null);
}
```

CastGroup.cs

```
public class CastGroup : IOrdered, IComparable, INamed, INotifyPropertyChanged,
IValidatable
{
    public CastGroup();

    [Key]
    public int CastGroupId { get; }
    public int Order { get; set; }
    public string Name { get; set; }
    public string Abbreviation { get; set; }
    public virtual ICollection<Applicant> Members { get; }
    public uint? RequiredCount { get; set; }
    public bool AlternateCasts { get; set; }
    public virtual ICollection<Requirement> Requirements { get; }

    public event PropertyChangedEventHandler? PropertyChanged;

    public uint FullTimeEquivalentMembers(int alternative_cast_count);
    [IteratorStateMachine(typeof(<Validate>d__38))]
```

```
    public IEnumerable<string> Validate();
    protected void OnPropertyChanged([CallerMemberName] string? name = null);
}
```

SameCastSet.cs

```
public class SameCastSet : INotifyPropertyChanged
{
    public SameCastSet();

    [Key]
    public int SameCastSetId { get; }
    public virtual ICollection<Applicant> Applicants { get; }
    public string Description { get; }

    public event PropertyChangedEventHandler? PropertyChanged;

    public bool VerifyAlternativeCasts(out AlternativeCast?
common_alternative_cast);
    protected void OnPropertyChanged([CallerMemberName] string? name = null);
}
```

Tag.cs

```
public class Tag : INameOrdered, INamed, IComparable, INotifyPropertyChanged,
IValidatable
{
    public Tag();

    [Key]
    public int TagId { get; }
    public string Name { get; set; }
    public string? Description { get; set; }
    public virtual Image? Icon { get; set; }
    public virtual ICollection<Applicant> Members { get; }
    public virtual ICollection<Requirement> Requirements { get; }
    public virtual ICollection<CountByGroup> CountByGroups { get; }

    public event PropertyChangedEventHandler? PropertyChanged;

    public uint? CountFor(CastGroup group);
    [IteratorStateMachine(typeof(<Validate>d__31))]
    public IEnumerable<string> Validate();
    protected void OnPropertyChanged([CallerMemberName] string? name = null);
}
```

Carmen.ShowModel.Criterias

BooleanCriteria.cs

```
public class BooleanCriteria : Criteria
{
    public BooleanCriteria();

    public override uint MaxMark { set; }

    public override string Format(uint mark);
}
```

Criteria.cs

```
public abstract class Criteria : IOrdered, IComparable, INamed,
INotifyPropertyChanged
{
    protected Criteria();

    [Key]
    public int CriteriaId { get; }
    public string Name { get; set; }
    public bool Required { get; set; }
    public bool Primary { get; set; }
    public int Order { get; set; }
    public double Weight { get; set; }
    public virtual ICollection<Ability> Abilities { get; }
    public virtual uint MaxMark { get; set; }

    public event PropertyChangedEventHandler? PropertyChanged;

    public abstract string Format(uint mark);
    protected void OnPropertyChanged([CallerMemberName] string? name = null);
}
```

NumericCriteria.cs

```
public class NumericCriteria : Criteria
{
    public NumericCriteria();

    public override string Format(uint mark);
}
```

SelectCriteria.cs

```
public class SelectCriteria : Criteria
{
    public SelectCriteria();

    public string[] Options { get; set; }
    public override uint MaxMark { set; }

    public override string Format(uint mark);
}
```

Carmen.ShowModel.Requirements

AbilityExactRequirement.cs

```
public class AbilityExactRequirement : ExactRequirement, ICriteriaRequirement,
IOrdered, IComparable, INamed
{
    public AbilityExactRequirement();

    public virtual Criteria Criteria { get; set; }
    public double ExistingRoleCost { get; set; }

    public override bool IsSatisfiedBy(Applicant applicant);
    [IteratorStateMachine(typeof(<Validate>d__13))]
    public override IEnumerable<string> Validate();
}
```

AbilityRangeRequirement.cs

```
public class AbilityRangeRequirement : RangeRequirement, ICriteriaRequirement,
IOrdered, IComparable, INamed
{
    public AbilityRangeRequirement();

    public virtual Criteria Criteria { get; set; }
    public bool ScaleSuitability { get; set; }
    public double ExistingRoleCost { get; set; }

    public override bool IsSatisfiedBy(Applicant applicant);
    [IteratorStateMachine(typeof(<Validate>d__17))]
    public override IEnumerable<string> Validate();
}
```

AgeRequirement.cs

```
public class AgeRequirement : RangeRequirement
{
    public AgeRequirement();

    public override bool IsSatisfiedBy(Applicant applicant);
}
```

CombinedRequirement.cs

```
public abstract class CombinedRequirement : Requirement
{
    public CombinedRequirement(params Requirement[] requirements);

    public virtual ICollection<Requirement> SubRequirements { get; }

    [IteratorStateMachine(typeof(<Validate>d__5))]
    public override IEnumerable<string> Validate();
}
public class AndRequirement : CombinedRequirement
{
    public AndRequirement();

    public bool AverageSuitability { get; set; }

    public override bool IsSatisfiedBy(Applicant applicant);
    public override bool IsSatisfiedBy(Applicant applicant,
ICollection<Requirement> accumulate_failed_requirements);
}
public class OrRequirement : CombinedRequirement
{
    public OrRequirement();

    public bool AverageSuitability { get; set; }

    public override bool IsSatisfiedBy(Applicant applicant);
}
public class XorRequirement : CombinedRequirement
{
    public XorRequirement();

    public override bool IsSatisfiedBy(Applicant applicant);
}
```

ExactRequirement.cs

```
public abstract class ExactRequirement : Requirement
{
    protected ExactRequirement();

    public uint RequiredValue { get; set; }
}
```

GenderRequirement.cs

```
public class GenderRequirement : ExactRequirement
{
    public GenderRequirement();

    public override bool IsSatisfiedBy(Applicant applicant);
}
```

NotRequirement.cs

```
public class NotRequirement : Requirement
{
    public NotRequirement();

    public virtual Requirement SubRequirement { get; set; }

    public override bool IsSatisfiedBy(Applicant applicant);
    [IteratorStateMachine(typeof(<Validate>d__10))]
    public override IEnumerable<string> Validate();
}
```

RangeRequirement.cs

```
public abstract class RangeRequirement : Requirement
{
    protected RangeRequirement();

    public uint? Minimum { get; set; }
    public uint? Maximum { get; set; }

    protected bool IsInRange(uint value);
}
```

Requirement.cs

```
public abstract class Requirement : IOverallWeighting, IOrdered, IComparable,
IValidatable, INamed, INotifyPropertyChanged
{
    protected Requirement();

    public int Order { get; set; }
    public virtual ICollection<CombinedRequirement> UsedByCombinedRequirements {
get; }
    public virtual ICollection<CastGroup> UsedByCastGroups { get; }
    public virtual ICollection<Role> UsedByRoles { get; }
    public string? Reason { get; set; }
    [Key]
    public int RequirementId { get; }
    public string Name { get; set; }
    public virtual ICollection<Tag> UsedByTags { get; }
    public double SuitabilityWeight { get; set; }
    public double OverallWeight { get; set; }
```

```
    public bool Primary { get; set; }

    public event PropertyChangedEventHandler? PropertyChanged;

    public virtual bool IsSatisfiedBy(Applicant applicant, ICollection<Requirement>
accumulate_failed_requirements);
    public abstract bool IsSatisfiedBy(Applicant applicant);
    public IEnumerable<Requirement> References();
    [IteratorStateMachine(typeof(<Validate>d__47))]
    public virtual IEnumerable<string> Validate();
    protected void OnPropertyChanged([CallerMemberName] string? name = null);
}
```

TagRequirement.cs

```
public class TagRequirement : Requirement
{
    public TagRequirement();

    public virtual Tag RequiredTag { get; set; }

    public override bool IsSatisfiedBy(Applicant applicant);
    [IteratorStateMachine(typeof(<Validate>d__9))]
    public override IEnumerable<string> Validate();
}
```

Carmen.ShowModel.Structure

ConsecutiveItemCast.cs

```
public struct ConsecutiveItemCast
{
    public Item Item1;
    public Item Item2;
    public HashSet<Applicant> Cast;
}
```

CountByGroup.cs

```
[Owned]
public class CountByGroup : INotifyPropertyChanged
{
    public CountByGroup();

    public virtual CastGroup CastGroup { get; set; }
    public uint Count { get; set; }

    public event PropertyChangedEventHandler? PropertyChanged;

    protected void OnPropertyChanged([CallerMemberName] string? name = null);
}
```

InnerNode.cs

```
public abstract class InnerNode : Node
{
    public InnerNode();

    public virtual ICollection<Node> Children { get; }
    public bool AllowConsecutiveItems { get; }
```

```
    protected abstract bool _allowConsecutiveItems { get; }

    public override IEnumerable<Item> ItemsInOrder();
    public bool VerifyConsecutiveItems(out List<ConsecutiveItemCast> failures);
    public bool VerifyConsecutiveItems();
}
```

Item.cs

```
public class Item : Node
{
    public Item();

    public virtual ICollection<Role> Roles { get; }

    public static HashSet<InnerNode> CommonParents(params Item[] items);
    [IteratorStateMachine(typeof(<FindConsecutiveCast>d__9))]
    public IEnumerable<ConsecutiveItemCast> FindConsecutiveCast();
    public IEnumerable<KeyValuePair<Applicant, int>> FindDuplicateCast();
    public override IEnumerable<Item> ItemsInOrder();
    public Item? NextItem();
    public Item? PreviousItem();
}
```

Node.cs

```
public abstract class Node : IOrdered, IComparable, ICounted, INamed,
INotifyPropertyChanged
{
    public Node();

    [Key]
    public int NodeId { get; }
    public string Name { get; set; }
    public int Order { get; set; }
    public virtual InnerNode? Parent { get; set; }
    public virtual ICollection<CountByGroup> CountByGroups { get; }

    public event PropertyChangedEventHandler? PropertyChanged;

    public uint CountFor(CastGroup group);
    public bool CountMatchesSumOfRoles();
    public abstract IEnumerable<Item> ItemsInOrder();
    [IteratorStateMachine(typeof(<Parents>d__27))]
    public IEnumerable<InnerNode> Parents();
    public Node? SiblingAbove();
    public Node? SiblingBelow();
    protected Dictionary<Applicant, int> CountRolesPerCastMember();
    protected void OnPropertyChanged([CallerMemberName] string? name = null);
}
```

Role.cs

```
public class Role : ICounted, INameOrdered, INamed, IComparable,
INotifyPropertyChanged
{
    public Role();

    [Key]
    public int RoleId { get; }
    public string Name { get; set; }
    public virtual ICollection<CountByGroup> CountByGroups { get; }
```

```
    public virtual ICollection<Item> Items { get; }
    public virtual ICollection<Requirement> Requirements { get; }
    public virtual ICollection<Applicant> Cast { get; }

    public event PropertyChangedEventHandler? PropertyChanged;

    public static HashSet<Node> CommonItemsAndSections(params Role[] roles);
    public RoleStatus CastingStatus(AlternativeCast[] alternative_casts);
    public uint CountFor(CastGroup group);
    public HashSet<Node> ItemsAndSections();
    protected void OnPropertyChanged([CallerMemberName] string? name = null);

    public enum RoleStatus
    {
        NotCast = 0,
        FullyCast = 1,
        OverCast = 2,
        UnderCast = 3
    }
}
```

Section.cs

```
public class Section : InnerNode
{
    public Section();

    public virtual SectionType SectionType { get; set; }
    protected override bool _allowConsecutiveItems { get; }

    public bool CastingMeetsSectionTypeRules(uint total_cast_members, out int
cast_with_no_roles, out int cast_with_multiple_roles);
    public bool CastingMeetsSectionTypeRules(IEnumerable<Applicant> cast_members,
out Applicant[] cast_with_no_roles, out KeyValuePair<Applicant, int>[]
cast_with_multiple_roles);
    public RolesMatchResult RolesMatchCastMembers(Dictionary<CastGroup, uint>
cast_members);

    public enum RolesMatchResult
    {
        RolesMatch = 0,
        TooFewRoles = 1,
        TooManyRoles = 2
    }
}
```

SectionType.cs

```
public class SectionType : INameOrdered, INamed, IComparable,
INotifyPropertyChanged
{
    public SectionType();

    [Key]
    public int SectionTypeId { get; }
    public string Name { get; set; }
    public bool AllowMultipleRoles { get; set; }
    public bool AllowNoRoles { get; set; }
    public bool AllowConsecutiveItems { get; set; }
    public virtual ICollection<Section> Sections { get; }

    public event PropertyChangedEventHandler? PropertyChanged;
```

```
    protected void OnPropertyChanged([CallerMemberName] string? name = null);
}
```

ShowRoot.cs

```
public class ShowRoot : InnerNode, IOverallWeighting
{
    public ShowRoot();

    public DateTime? ShowDate { get; set; }
    public virtual Image? Logo { get; set; }
    public virtual Criteria? CastNumberOrderBy { get; set; }
    public ListSortDirection CastNumberOrderDirection { get; set; }
    public bool AllowConsecutiveItems { get; set; }
    public double? CommonOverallWeight { get; set; }
    public bool WeightExistingRoleCosts { get; set; }
    public override InnerNode? Parent { get; set; }
    protected override bool _allowConsecutiveItems { get; }
}
```

Appendix D - Casting engine interfaces

The critical operations performed within the CARMEN application are abstracted into 3 engine interfaces which manipulate the ShowModel. These engine interfaces are included below.

IAllocationEngine.cs

```
// Interface for CastingEngine functions relating to Role allocation
public interface IAllocationEngine
{
    // An accessor to the IAuditionEngine used by this allocation engine
    IAuditionEngine AuditionEngine { get; }

    // Determine the recommended order in which the roles should be cast. Each
    // array in the sequence should be cast as one step, ie. a single element array
    // recommends a call to PickCast(IEnumerable<Applicant>, Role),
    // whereas a multi-element array recommends a call to
    // BalanceCast(IEnumerable<Applicant>, IEnumerable<Role>).
    // This should return all roles in the show exactly once, regardless of
    // whether or not they are already cast.
    IEnumerable<Role[]> IdealCastingOrder(ShowRoot show_root, Applicant[]
applicants_in_cast);

    // Calculate the suitability of an applicant for a role, regardless of
    // availability and eligibility. Value returned will be between 0 and 1
    // (inclusive). This may contain logic specific to roles, and is therefore
    // different to IAuditionEngine.SuitabilityOf(Applicant, Requirement).
    double SuitabilityOf(Applicant applicant, Role role);

    // Count the number of roles an applicant has which require a certain criteria,
    // optionally excluding a specified role. Depending on implementation, this
    // may return a whole number of actual roles, or a sum of prorated roles based
    // on the importance of the specified criteria to each role.
    double CountRoles(Applicant applicant, Criteria criteria, Role?
excluding_role);

    // Pick the best cast for a role, returning the chosen cast without casting
    // them.
    IEnumerable<Applicant> PickCast(IEnumerable<Applicant> applicants, Role role);

    // A callback for when the user allocates cast to a role, providing information
    // to the engine which can be used to improve future recommendations.
    void UserPickedCast(IEnumerable<Applicant> applicants_picked,
IEnumerable<Applicant> applicants_not_picked, Role role);

    // Saves any changes made in the engine back to the ShowModel or otherwise.
    // This should be called whenever the engine's job is finished.
    void ExportChanges();

    // Allocate the best cast to one or more roles, balancing talent between them.
    // NOTE: Unlike PickCast(IEnumerable<Applicant>, Role) this directly allocates
    // cast to the role rather than returning the chosen cast.
    void BalanceCast(IEnumerable<Applicant> applicants, IEnumerable<Role> roles);

    // Determine if an applicant is eligible to be cast in a role
    // (ie. whether all minimum requirements of the role are met)
    Eligibility EligibilityOf(Applicant applicant, Role role);

    // Determine if an applicant is available to be cast in a role
    // (eg. already cast in the same item, an adjacent item, or within a section
```

```
    // where AllowMultipleRoles==FALSE)
    Availability AvailabilityOf(Applicant applicant, Role role);
}
```

IAuditionEngine.cs

```
// Interface for CastingEngine functions relating to auditioning Applicants
public interface IAuditionEngine
{
    // The maximum value an applicant's overall ability can be
    int MaxOverallAbility { get; }

    // The minimum value an applicant's overall ability can be
    int MinOverallAbility { get; }

    // Calculate the overall ability of an applicant
    int OverallAbility(Applicant applicant);

    // A callback for when the user selects cast into cast groups manually,
    // providing information to the engine which can be used to improve future
    // recommendations.
    void UserSelectedCast(IEnumerable<Applicant> applicants_accepted,
IEnumerable<Applicant> applicants_rejected);

    // Calculate the suitability of an applicant against a single requirement.
    // Value returned will be between 0 and 1 (inclusive).
    double SuitabilityOf(Applicant applicant, Requirement requirement);

    // Calculate the overall ability of an applicant as a suitability value
    // between 0 and 1 (inclusive).
    double OverallSuitability(Applicant applicant);
}
```

ISelectionEngine.cs

```
// Interface for CastingEngine functions relating to Cast selection
public interface ISelectionEngine
{
    // An accessor to the IAuditionEngine used by this selection engine
    IAuditionEngine AuditionEngine { get; }

    // Calculate the suitability of an applicant for a cast group, regardless of
    // whether they meet all requirements. Value returned will be between 0 and 1
    // (inclusive). This may contain logic specific to cast groups, and is
    // therefore different to IAuditionEngine.SuitabilityOf(Applicant, Requirement)
    double SuitabilityOf(Applicant applicant, CastGroup cast_group);

    // Calculate the suitability of an applicant for a tag, regardless of whether
    // they meet all requirements. Value returned will be between 0 and 1
    // (inclusive). This may contain logic specific to tags, and is therefore
    // different to IAuditionEngine.SuitabilityOf(Applicant, Requirement).
    double SuitabilityOf(Applicant applicant, Tag tag);

    // Set same cast sets for family groups within the applicants provided.
    void DetectFamilies(IEnumerable<Applicant> applicants, out List<SameCastSet>
new_same_cast_sets);

    // Select applicants into cast groups, respecting those already placed
    // NOTE: CastGroup requirements may not depend on Tags
    void SelectCastGroups(IEnumerable<Applicant> applicants, IEnumerable<CastGroup>
cast_groups);
```

```
    // Balance applicants between alternative casts, respecting those already set
    // NOTE: This will clear the AlternativeCast of rejected Applicants
    void BalanceAlternativeCasts(IEnumerable<Applicant> applicants,
IEnumerable<SameCastSet> same_cast_sets);

    // Allocate cast numbers, respecting those already set, ordered by a criteria,
    // otherwise overall ability
    // NOTE:
    // - This will clear the cast number of rejected Applicants
    // - Accepted applicants must have an AlternativeCast set when
    //   CastGroup.AlternateCasts == true
    void AllocateCastNumbers(IEnumerable<Applicant> applicants);

    // Apply tags to applicants, respecting those already applied
    // NOTE:
    // - This will remove Tags from rejected Applicants
    // - Accepted applicants must have an AlternativeCast set when
    //   CastGroup.AlternateCasts == true
    // - Tag requirements may depend on other Tags as long as there is no circular
    //   dependency and the dependee tag is also being applied as part of this call
    void ApplyTags(IEnumerable<Applicant> applicants, IEnumerable<Tag> tags);
}
```

Appendix E - Heuristic engine implementations

The relevant functions of heuristic engine implementations are included in this appendix. These are used as a baseline on which to measure the improvement of SAT solving (Appendix F - SAT solving ISelectionEngine implementations) and neural network based (Appendix H - Neural network IAllocationEngine implementations) implementations.

HeuristicAllocationEngine.cs

```
public class HeuristicAllocationEngine : AllocationEngine
{
    readonly Criteria[] criterias;

    public HeuristicAllocationEngine(IAuditionEngine audition_engine,
AlternativeCast[] alternative_casts, Criteria[] criterias)
        : base(audition_engine, alternative_casts)
    {
        CountRolesByGeometricMean = false;
        CountRolesIncludingPartialRequirements = false;
        this.criterias = criterias;
    }

    // Essentially this is the same as WeightedAverageEngine, except with
    // hardcoded values:
    // - OverallSuitabilityWeight is always 1
    // - Each criteria requirement's SuitabilityWeight is 2
    // - Each criteria requirement's ExistingRoleCost is 0.5
    // - Only direct criteria requirements are included
    public override double SuitabilityOf(Applicant applicant, Role role)
    {
        double score = AuditionEngine.OverallSuitability(applicant); // between 0
and 1 inclusive
        var max = 1;
        foreach (var requirement in role.Requirements)
        {
            if (requirement is ICriteriaRequirement based_on)
            {
                score += 2 * AuditionEngine.SuitabilityOf(applicant, requirement) -
0.5 * CountRoles(applicant, based_on.Criteria, role) / 100.0;
                max += 2;
            }
        }
        return score / max;
    }

    public IEnumerable<Role[]> SimpleCastingOrder(ShowRoot show_root);

    private IEnumerable<InnerNode> ItemContainingSections(InnerNode inner);
}
```

HeuristicSelectionEngine.cs

```
public class HeuristicSelectionEngine : SelectionEngine
{
    public HeuristicSelectionEngine(IAuditionEngine audition_engine,
AlternativeCast[] alternative_casts, Criteria? cast_number_order_by,
ListSortDirection cast_number_order_direction);
```

```
    // Allocate applicants into alternative casts by grouping by cast group,
    // counting the applicants already in casts, allocating the same_cast_set
    // applicants first, then filling the rest by sorting the applicants into cast
    // number order and putting one at a time into the cast with the current
    // smallest count.
    public override void BalanceAlternativeCasts(IEnumerable<Applicant> applicants,
IEnumerable<SameCastSet> same_cast_sets)
    {
        same_cast_sets = same_cast_sets.ToArray(); // make it safe to enumerate
repeatedly
        // sort the applicants into cast groups
        foreach (var applicants_group in applicants.GroupBy(a => a.CastGroup))
        {
            if (applicants_group.Key is not CastGroup cast_group // not accepted
into cast
                || !cast_group.AlternateCasts) // non-alternating cast group
            {
                // alternative cast not required, set to null
                foreach (var applicant in applicants_group)
                    applicant.AlternativeCast = null;
            }
            else
            {
                // count existing alternative casts
                var applicants_needing_cast = new List<Applicant>();
                var cast_counts = new int[alternativeCasts.Length];
                foreach (var applicant in applicants_group)
                {
                    if (applicant.AlternativeCast is AlternativeCast ac)
                    {
                        var index = Array.IndexOf(alternativeCasts, ac);
                        if (index < 0)
                            throw new ApplicationException($"Alternative cast
'{ac.Name}' not found in AlternativeCasts provided.");
                        cast_counts[index]++;
                    }
                    else
                        applicants_needing_cast.Add(applicant);
                }
                // allocate casts to same cast sets first
                foreach (var same_cast_set in same_cast_sets)
                {
                    int cast_index;
                    if (same_cast_set.Applicants.Select(a =>
a.AlternativeCast).OfType<AlternativeCast>().Distinct().SingleOrDefaultSafe() is
AlternativeCast alternative_cast)
                        cast_index = Array.IndexOf(alternativeCasts,
alternative_cast);
                    else
                        cast_index = MinimumIndex(cast_counts);
                    foreach (var unset_cast in same_cast_set.Applicants.Where(a =>
applicants_needing_cast.Contains(a)))
                    {
                        unset_cast.AlternativeCast = alternativeCasts[cast_index];
                        cast_counts[cast_index]++;
                        applicants_needing_cast.Remove(unset_cast);
                    }
                }
                // allocate remaining alternative casts
                int next_cast = 0;
                foreach (var applicant in
CastNumberingOrder(applicants_needing_cast))
                {
```

```
                    applicant.AlternativeCast = alternativeCasts[next_cast++];
                    if (next_cast == alternativeCasts.Length)
                        next_cast = 0;
                }
            }
        }
    }

    // Finds the index of the minimum value in the array
    private int MinimumIndex(int[] counts);
}
```

WeightedSumEngine.cs

```
// A basic AuditionEngine which calculates an Applicant's overall ability as a
// weighted sum of their abilities.
public class WeightedSumEngine : AuditionEngine
{
    int maxOverallAbility;
    public override int MaxOverallAbility => maxOverallAbility;

    int minOverallAbility;
    public override int MinOverallAbility => minOverallAbility;

    public WeightedSumEngine(Criteria[] criterias)
    {
        UpdateRange(criterias);
    }

    readonly FunctionCache<Applicant, int> overallAbility = new(applicant
        => Convert.ToInt32(applicant.Abilities.Sum(a => (double)a.Mark /
a.Criteria.MaxMark * a.Criteria.Weight)));

    // Calculate the overall ability of an Applicant as a weighted sum of their
    // Abilities. NOTE: This is cached for speed, as an Applicant's abilities
    // shouldn't change over the lifetime of an AuditionEngine
    public override int OverallAbility(Applicant applicant) =>
overallAbility[applicant];

    protected void UpdateRange(Criteria[] criterias)
    {
        var max = criterias.Select(c => c.Weight).Where(w => w > 0).Sum();
        if (max > int.MaxValue)
            throw new ApplicationException($"Sum of positive Criteria weights
cannot exceed {int.MaxValue}: {max}");
        maxOverallAbility = Convert.ToInt32(max);
        var min = criterias.Select(c => c.Weight).Where(w => w < 0).Sum();
        if (min < int.MinValue)
            throw new ApplicationException($"Sum of negative Criteria weights
cannot go below {int.MinValue}: {min}");
        minOverallAbility = Convert.ToInt32(min);
        if (minOverallAbility == maxOverallAbility) // == 0
            maxOverallAbility = 1; // to avoid division by zero errors
    }
}
```

Appendix F - SAT solving ISelectionEngine implementations

The relevant functions of SAT solving engine implementations are included in this appendix. These are compared to the heuristic implementations in Appendix E - Heuristic engine implementations.

BestPairsSatEngine.cs

```
// A concrete approach for balancing alternative casts based on TopPairsSatEngine,
// but using a Branch & Bound algorithm to minimize the rank difference.
public class BestPairsSatEngine : TopPairsSatEngine
{
    int[][] applicantRanks = Array.Empty<int[]>(); // [criteria][applicant
variable]
    Applicant[] applicantVariables = Array.Empty<Applicant>();
    int[] castGroupIndexFromVariableIndex = Array.Empty<int>();
    CastGroup[] castGroups = Array.Empty<CastGroup>();

    public BestPairsSatEngine(IAuditionEngine audition_engine, AlternativeCast[]
alternative_casts, Criteria? cast_number_order_by, ListSortDirection
cast_number_order_direction, Criteria[] criterias);

    protected override Solver<Applicant> BuildSatSolver(List<(CastGroup,
HashSet<Applicant>)> applicants_needing_alternative_cast)
    {
        RankDifferenceSatEngine.CalculateRankings(primaryCriterias,
applicants_needing_alternative_cast, out applicantVariables, out castGroups, out
applicantRanks, out castGroupIndexFromVariableIndex);
        return new BranchAndBoundSolver<Applicant>(CostFunction,
applicantVariables)
        {
            StagnantTimeout = RankDifferenceSatEngine.TIMEOUT_MS
        };
    }

    private (double lower, double upper) CostFunction(Solution partial_solution)
        => RankDifferenceSatEngine.RankDifference(partial_solution,
primaryCriterias, castGroups, applicantRanks, castGroupIndexFromVariableIndex);
}
```

ChunkedPairsSatEngine.cs

```
// A concrete approach for balancing alternative casts by initially pairing off
// the full list of applicants in order, then increasing from pairs to a larger
// chunk size each iteration until success.
public class ChunkedPairsSatEngine : PairsSatEngine
{
    public ChunkedPairsSatEngine(IAuditionEngine audition_engine, AlternativeCast[]
alternative_casts, Criteria? cast_number_order_by, ListSortDirection
cast_number_order_direction, Criteria[] criterias);

    protected override bool Simplify(ref int chunk_size, ref int max_chunks)
    {
        chunk_size += 2;
        return chunk_size <= 16; // any more than 16 takes too long to process
    }
}
```

HybridPairsSatEngine.cs

```
// A concrete approach for balancing alternative casts with a combination of the
// TopPairs and ChunkedPairs approaches. Initially the full list of aplicants are
// paired off in order, reducing the number of pairs each iteration until success.
// The remaining applicants are then chunked, increasing the chunk size until it
// succeeds.
public class HybridPairsSatEngine : PairsSatEngine
{
    public HybridPairsSatEngine(IAuditionEngine audition_engine, AlternativeCast[]
alternative_casts, Criteria? cast_number_order_by, ListSortDirection
cast_number_order_direction, Criteria[] criterias);

    protected override void Initialise(out int chunk_size, out int max_chunks)
    {
        chunk_size = 2;
        max_chunks = int.MaxValue;
    }

    protected override bool Simplify(ref int chunk_size, ref int max_chunks)
    {
        max_chunks -= 1;
        return max_chunks >= 0; // allow zero in case the basic clauses are
solvable on their own
    }

    protected override bool Improve(ref int chunk_size, ref int max_chunks)
    {
        chunk_size += 2;
        max_chunks = int.MaxValue; // usually terminates when no additional chunk
clauses are added
        return chunk_size <= 16; // any more than 16 takes too long to process
    }
}
```

PairsSatEngine.cs

```
// A type of approach for balancing alternative casts which uses a DPLL solver for
// the k-SAT boolean satisfiability problem, modelling each Applicant's choice
// between 2 alternative casts as a boolean variable.
// NOTE: This only works for exactly 2 alternative casts
public abstract class PairsSatEngine : SatEngine
{
    public List<Result> Results { get; init; } = new();

    public struct Result { ... }

    // expression building is O(2^n) and will be slow for large numbers, therefore
    // cache
    Dictionary<int, ExpressionBuilder> cachedKeepSeparate = new();

    public PairsSatEngine(IAuditionEngine audition_engine, AlternativeCast[]
alternative_casts, Criteria? cast_number_order_by, ListSortDirection
cast_number_order_direction, Criteria[] criterias);

    protected override Solver<Applicant> BuildSatSolver(List<(CastGroup,
HashSet<Applicant>)> applicants_needing_alternative_cast)
        => new
DpllSolver<Applicant>(applicants_needing_alternative_cast.SelectMany(p =>
p.Item2));

    protected override Solution FindSatSolution(Solver<Applicant> sat,
List<(CastGroup, HashSet<Applicant>)> applicants_needing_alternative_cast,
```

```
        List<Clause<Applicant>> existing_assignments, List<Clause<Applicant>>
same_cast_clauses, Dictionary<Applicant, SameCastSet> same_cast_lookup)
    {
        // Initialise() the starting conditions, then Simplify() until the SAT
solver succeeds
        var i = 0;
        var applicant_variable_numbers = sat.Variables.ToDictionary(v => v, v =>
i++);
        Solution solution = Solution.Unsolveable;
        var solved_clauses = new HashSet<Clause<int>>();
        solved_clauses.AddRange(existing_assignments.Select(c =>
c.Remap(applicant_variable_numbers)));
        solved_clauses.AddRange(same_cast_clauses.Select(c =>
c.Remap(applicant_variable_numbers)));
        int previously_chunked_applicants = 0;
        Results.Clear();
        Initialise(out int chunk_size, out int max_chunks);
        do
        {
            // cap max_chunk to the highest possible number of chunks
            var max_possible_chunks = MaximumPossibleChunks(chunk_size,
previously_chunked_applicants, applicants_needing_alternative_cast.Select(p =>
p.Item2.Count));
            if (max_chunks > max_possible_chunks)
                max_chunks = max_possible_chunks;
            // compile clauses
            var clauses = new HashSet<Clause<int>>(solved_clauses);
            var max_actual_chunks = 0;
            foreach (var (cg, hs) in applicants_needing_alternative_cast)
            {
                AddChunkClauses(clauses, hs, previously_chunked_applicants,
chunk_size, max_chunks, same_cast_lookup, out var actual_chunks,
applicant_variable_numbers);
                if (actual_chunks > max_actual_chunks)
                    max_actual_chunks = actual_chunks;
            }
            if (clauses.Count == 0)
                break; // no clauses to solve
            // run sat solver
            solution = sat.SolveWithoutRemap(new(clauses)).FirstOrDefault();
            Results.Add(new Result
            {
                ChunkSize = chunk_size,
                MaxChunks = max_chunks,
                Variables = sat.Variables.Count,
                Clauses = clauses.Count,
                MaxLiterals = clauses.Max(c => c.Literals.Count),
                Solved = !solution.IsUnsolvable
            });
            if (!solution.IsUnsolvable)
            {
                previously_chunked_applicants += chunk_size * max_actual_chunks; //
using the maximum number of actual chunks across all cast groups only works if
chunk_size is only ever increased
                if (Improve(ref chunk_size, ref max_chunks))
                    solved_clauses.AddRange(clauses);
                else
                    break; // solved, and no further improvements
            }
            if (max_actual_chunks == 0)
                break; // we didn't have any chunk clauses, if this didn't solve,
nothing will
        } while (Simplify(ref chunk_size, ref max_chunks));
```

```
        return solution;
    }

    // This is called to initialise the chunking parameters before we start SAT
    // solving. The default implementation is to start by chunking the whole list
    // of applicants into pairs. It may be helpful to override this if your
    // Simplify() method uses state which needs to be reset between runs.
    protected virtual void Initialise(out int chunk_size, out int max_chunks)
    {
        chunk_size = 2;
        max_chunks = int.MaxValue;
    }

    // If the current SAT problem is unsolvable, this is called to modify the
    // chunking parameters to make it a simplier SAT problem to solve. If there is
    // no simplier option, return false to stop trying. As a failsafe, if the
    // chunking parameters fail to create any chunks, iteration will terminate.
    protected abstract bool Simplify(ref int chunk_size, ref int max_chunks);

    // If the current SAT problem is solveable, this is called to modify the
    // chunking parameters to make it a harder SAT problem to solve, that is, one
    // which would produce a better outcome. In this case, the existing clauses
    // are kept in addition to any new clauses created. If there is no harder
    // option, return false to accept the current solution. The default
    // implementation always accepts the current solution.
    protected virtual bool Improve(ref int chunk_size, ref int max_chunks) =>
false;

    // Find the value of max_chunks at which the limit will not be hit before all
    // applicants have been chunked
    private int MaximumPossibleChunks(int chunk_size, int previously_chunked,
IEnumerable<int> applicants_per_cast_group)
        => (applicants_per_cast_group.Max() - previously_chunked) / chunk_size;

    // Creates clauses for chunks of applicants within one cast group, for each
    // primary criteria, and add them to the supplied collection of clauses
    private void AddChunkClauses(ICollection<Clause<int>> clauses,
IEnumerable<Applicant> applicants, int skip_applicants, int chunk_size, int
max_chunks,
        Dictionary<Applicant, SameCastSet> same_cast_lookup, out int
max_chunk_count, Dictionary<Applicant, int> applicant_variable_numbers)
    {
        max_chunk_count = 0;
        foreach (var criteria in primaryCriterias)
        {
            int chunk_count = 0;
            var sorted_applicants = new Stack<Applicant>(applicants.OrderBy(a =>
a.MarkFor(criteria))); // order by marks ascending so that the lowest mark is at
the bottom of the stack
            while (skip_applicants-- > 0 && sorted_applicants.Any())
                sorted_applicants.Pop();
            while (sorted_applicants.Count >= chunk_size && chunk_count <
max_chunks)
            {
                if (TakeChunk(sorted_applicants, chunk_size, same_cast_lookup) is
Applicant[] chunk)
                    foreach (var clause in KeepSeparate(chunk.Select(a =>
applicant_variable_numbers[a]).ToArray()))
                        clauses.Add(clause);
                else
                    // TakeChunk may fail, even though there are technically enough
applicants,
                    // if the remaining applicants are in a SameCastSet
```

```
                    break;
                chunk_count++;
            }
            if (chunk_count > max_chunk_count)
                max_chunk_count = chunk_count;
        }
    }

    // Creates a clause specifying that exactly half of the given applicants are
    // in each alternative cast
    private IEnumerable<Clause<int>> KeepSeparate(int[]
applicant_variable_numbers);

    // Takes chunk_size applicants from the top of the stack, with at most
    // chunk_size/2 from any one SameCastSet
    private Applicant[]? TakeChunk(Stack<Applicant> applicants, int chunk_size,
Dictionary<Applicant, SameCastSet> same_cast_lookup);
}
```

RankDifferenceSatEngine.cs

```
// A concrete approach for balancing alternative casts by minimising difference of
// the sum of ranks for each criteria, between casts.
// NOTE: This only works for exactly 2 alternative casts
public class RankDifferenceSatEngine : SatEngine
{
    public const int RANK_INCREMENT = 2;
    public const int TIMEOUT_MS = 5000;

    int[][] applicantRanks = Array.Empty<int[]>(); // [criteria][applicant
variable]
    Applicant[] applicantVariables = Array.Empty<Applicant>();
    int[] castGroupIndexFromVariableIndex = Array.Empty<int>();
    CastGroup[] castGroups = Array.Empty<CastGroup>();

    public RankDifferenceSatEngine(IAuditionEngine audition_engine,
AlternativeCast[] alternative_casts, Criteria? cast_number_order_by,
ListSortDirection cast_number_order_direction, Criteria[] criterias);

    protected override Solver<Applicant> BuildSatSolver(List<(CastGroup,
HashSet<Applicant>)> applicants_needing_alternative_cast)
    {
        CalculateRankings(primaryCriterias, applicants_needing_alternative_cast,
out applicantVariables, out castGroups, out applicantRanks, out
castGroupIndexFromVariableIndex);
        var success_threshold = RANK_INCREMENT * primaryCriterias.Length *
castGroups.Length; // 1 rank difference per criteria per cast group
        return new BranchAndBoundSolver<Applicant>(CostFunction,
applicantVariables)
        {
            SuccessThreshold = success_threshold,
            StagnantTimeout = TIMEOUT_MS
        };
    }

    protected override Solution FindSatSolution(Solver<Applicant> sat,
List<(CastGroup, HashSet<Applicant>)> applicants_needing_alternative_cast,
        List<Clause<Applicant>> existing_assignments, List<Clause<Applicant>>
same_cast_clauses, Dictionary<Applicant, SameCastSet> same_cast_lookup)
    {
        var clauses = new HashSet<Clause<Applicant>>();
        clauses.AddRange(existing_assignments);
```

```
        clauses.AddRange(same_cast_clauses);
        if (clauses.Count == 0)
        {
            if (sat.Variables.Count == 0)
                return Solution.Unsolveable;
            var first_var = sat.Variables.First();
            clauses.Add(new(new[] {
                Literal<Applicant>.Positive(first_var),
                Literal<Applicant>.Negative(first_var)
            }.ToHashSet()));
        }
        var solution = sat.Solve(new(clauses)).FirstOrDefault();
        return solution;
    }

    internal static void CalculateRankings(Criteria[] primaryCriterias,
List<(CastGroup, HashSet<Applicant>)> applicants_needing_alternative_cast,
        out Applicant[] applicantVariables, out CastGroup[] castGroups, out int[][]
applicantRanks, out int[] castGroupIndexFromVariableIndex)
    {
        applicantVariables = applicants_needing_alternative_cast.SelectMany(p =>
p.Item2).ToArray();
        castGroups = applicantVariables.Select(a =>
a.CastGroup!).Distinct().ToArray();
        applicantRanks = new int[primaryCriterias.Length][];
        castGroupIndexFromVariableIndex = new int[applicantVariables.Length];
        for (var c = 0; c < primaryCriterias.Length; c++)
        {
            applicantRanks[c] = new int[applicantVariables.Length];
            foreach (var (cast_group, applicants) in
applicants_needing_alternative_cast)
            {
                var cg = Array.IndexOf(castGroups, cast_group);
                int rank = 0;
                uint? previous = null;
                foreach (var (applicant, mark) in applicants.Select(a => (a,
a.MarkFor(primaryCriterias[c]))).OrderBy(p => p.Item2))
                {
                    var av = Array.IndexOf(applicantVariables, applicant);
                    castGroupIndexFromVariableIndex[av] = cg;
                    if (mark != previous)
                    {
                        applicantRanks[c][av] = rank;
                        rank += RANK_INCREMENT;
                        previous = mark;
                    }
                    else
                        applicantRanks[c][av] = rank;
                }
            }
        }
    }

    private (double lower, double upper) CostFunction(Solution partial_solution)
        => RankDifference(partial_solution, primaryCriterias, castGroups,
applicantRanks, castGroupIndexFromVariableIndex);

    // Calculates the bounds of the rank difference for a given partial solution.
    // Returns double.MaxValue for both if the solution will not have even casts.
    internal static (double lower, double upper) RankDifference(Solution
partial_solution, Criteria[] primaryCriterias, CastGroup[] castGroups, int[][]
applicantRanks, int[] castGroupIndexFromVariableIndex)
    {
```

```
        var best_total = 0;
        var worst_total = 0;
        for (var c = 0; c < primaryCriterias.Length; c++)
        {
            var assigned_rank_difference = new int[castGroups.Length];
            var not_assigned_ranks = new List<int>[castGroups.Length];
            for (var cg = 0; cg < castGroups.Length; cg++)
                not_assigned_ranks[cg] = new List<int>();
            var count_true = new int[castGroups.Length];
            var count_false = new int[castGroups.Length];
            var count_total = new int[castGroups.Length];
            for (var av = 0; av < partial_solution.Assignments.Length; av++)
            {
                var cg = castGroupIndexFromVariableIndex[av];
                var rank = applicantRanks[c][av];
                if (partial_solution.Assignments[av] is not bool value)
                    not_assigned_ranks[cg].Add(rank);
                else if (value)
                {
                    assigned_rank_difference[cg] += rank;
                    count_true[cg]++;
                }
                else
                {
                    assigned_rank_difference[cg] -= rank;
                    count_false[cg]++;
                }
                count_total[cg]++;
            }
            var best_criteria = 0;
            var worst_criteria = 0;
            for (var cg = 0; cg < castGroups.Length; cg++)
            {
                int max = (count_total[cg] + 1) / 2;
                if (count_true[cg] > max || count_false[cg] > max)
                    return (double.MaxValue, double.MaxValue); // invalid solution,
casts are not even
                var not_assigned_sorted =
not_assigned_ranks[cg].OrderByDescending(r => r).ToArray();
                var assignable_true = not_assigned_sorted.Take(max -
count_true[cg]).Sum();
                var assignable_false = not_assigned_sorted.Take(max -
count_false[cg]).Sum();
                var (best_cg, worst_cg) =
FindBestAndWorstCases(assigned_rank_difference[cg], assignable_true,
assignable_false);
                best_criteria += best_cg;
                worst_criteria += worst_cg;
            }
            best_total += best_criteria;
            worst_total += worst_criteria;
        }
        return (best_total, worst_total);
    }

    private static (int best, int worst) FindBestAndWorstCases(int current, int
max_add, int max_subtract)
    {
        int best_case;
        if (current > 0 && max_subtract < current)
            best_case = current - max_subtract;
        else if (current < 0 && max_add < -current)
            best_case = -current - max_add;
```

```
        else
            best_case = 0;
        int worst_case = Math.Max(Math.Abs(current + max_add), Math.Abs(current -
max_subtract));
        return (best_case, worst_case);
    }
}
```

SatEngine.cs

```
// The base class of all SAT based selection engines
public abstract class SatEngine : SelectionEngine
{
    protected Criteria[] primaryCriterias;

    // expression building is O(2^n) and will be slow for large numbers, therefore
    // cache
    Dictionary<int, ExpressionBuilder> cachedKeepTogether = new();

    public SatEngine(IAuditionEngine audition_engine, AlternativeCast[]
alternative_casts, Criteria? cast_number_order_by, ListSortDirection
cast_number_order_direction, Criteria[] criterias);

    protected abstract Solver<Applicant> BuildSatSolver(List<(CastGroup,
HashSet<Applicant>)> applicants_needing_alternative_cast);

    protected abstract Solution FindSatSolution(Solver<Applicant> sat,
List<(CastGroup, HashSet<Applicant>)> applicants_needing_alternative_cast,
        List<Clause<Applicant>> existing_assignments, List<Clause<Applicant>>
same_cast_clauses, Dictionary<Applicant, SameCastSet> same_cast_lookup);

    public override void BalanceAlternativeCasts(IEnumerable<Applicant> applicants,
IEnumerable<SameCastSet> same_cast_sets)
    {
        if (alternativeCasts.Length != 2)
            throw new ApplicationException("This approach only works for exactly 2
alternative casts.");
        // sort the applicants into cast groups
        var applicants_needing_alternative_cast = new List<(CastGroup,
HashSet<Applicant>)>();
        var existing_assignments = new List<Clause<Applicant>>();
        foreach (var applicants_group in applicants.GroupBy(a => a.CastGroup))
        {
            if (applicants_group.Key is not CastGroup cast_group // not accepted
into cast
                || !cast_group.AlternateCasts) // non-alternating cast group
            {
                // alternative cast not required, set to null
                foreach (var applicant in applicants_group)
                    applicant.AlternativeCast = null;
            }
            else
            {
                var applicants_set = applicants_group.ToHashSet();
                applicants_needing_alternative_cast.Add((cast_group,
applicants_set));
                // add clauses for applicants with alternative cast already set
                foreach (var applicant in applicants_set)
                    if (applicant.AlternativeCast != null)
                        existing_assignments.Add(Clause<Applicant>.Unit(applicant,
applicant.AlternativeCast == alternativeCasts[1]));
            }
```

```
        }
        if (applicants_needing_alternative_cast.Count == 0)
            return; // nothing to do
        // create clauses for same cast sets
        var same_cast_clauses = new List<Clause<Applicant>>();
        var same_cast_lookup = new Dictionary<Applicant, SameCastSet>();
        foreach (var same_cast_set in same_cast_sets)
        {
            foreach (var applicant in same_cast_set.Applicants)
                same_cast_lookup.Add(applicant, same_cast_set); // Applicants can
only be in one SameCastSet

same_cast_clauses.AddRange(KeepTogether(same_cast_set.Applicants.Where(a =>
applicants_needing_alternative_cast.Any(p => p.Item2.Contains(a))).ToArray()));
        }
        // pre-test clauses to ensure they are solvable
        var sat = BuildSatSolver(applicants_needing_alternative_cast);
        var base_expression = new
Expression<Applicant>(existing_assignments.Concat(same_cast_clauses).ToHashSet());
        var solution = sat.Solve(base_expression).FirstOrDefault();
        // run approach-specific SAT solving
        if (!solution.IsUnsolvable)
            solution = FindSatSolution(sat, applicants_needing_alternative_cast,
existing_assignments, same_cast_clauses, same_cast_lookup);
        // set alternative casts
        List<IEnumerable<Assignment<Applicant>>> grouped_assignments;
        if (!solution.IsUnsolvable)
            // assigned by sat solver
            grouped_assignments = sat.MapAssignments(solution)
                .GroupBy(a => a.Variable.CastGroup)
                .Select(g => g.AsEnumerable()).ToList();
        else
            // default to applicants evenly filled
            grouped_assignments = applicants_needing_alternative_cast
                .Select(p => p.Item2.Select(a => new Assignment<Applicant>
                {
                    Variable = a,
                    Value = a.AlternativeCast?.Equals(alternativeCasts[1]) // keep
the existing value if set
                })).ToList();
        foreach (var group in grouped_assignments)
        {
            var full_assignments = EvenlyFillAssignments(group);
            foreach (var assignment in full_assignments)
                assignment.Variable.AlternativeCast =
alternativeCasts[assignment.Value!.Value ? 1 : 0];
        }
    }

    // Creates a clause specifying that all these applicants are in the same
    // alternative cast
    private IEnumerable<Clause<Applicant>> KeepTogether(Applicant[] applicants);

    // Fills any free (null) assignments, keeping the totals in each alternative
    // cast as close to equal as possible
    private Assignment<Applicant>[]
EvenlyFillAssignments(IEnumerable<Assignment<Applicant>> assignments);
}
```

ThreesACrowdSatEngine.cs

```
// A concrete approach for balancing alternative casts which uses a DPLL(T) solver
```

```
// for the SMT satisfiability problem, by creating overlapping sets of 3 applicants
// which must not all be in the same cast. The total number of applicants in each
// cast is ensured by the validity theory test. If unsolvable, the number of sets
// is reduced until it is.
// NOTE: This currently only works for exactly 2 alternative casts
public class ThreesACrowdSatEngine : SatEngine
{
    public List<Result> Results { get; init; } = new();

    public struct Result { ... }

    public ThreesACrowdSatEngine(IAuditionEngine audition_engine, AlternativeCast[]
alternative_casts, Criteria? cast_number_order_by, ListSortDirection
cast_number_order_direction, Criteria[] criterias);

    // Verifies that the number of cast assigned to each alternative cast is the
    // same for all cast groups. If the number of cast in a cast group is odd,
    // the spare cast member can be either alternative cast.
    private bool? AlternativeCastsAreEqual(bool?[] assignments, CastGroup[]
cast_groups)
    {
        if (assignments.Length != cast_groups.Length)
            throw new ArgumentException("Assignments and cast groups must have the
same number of elements");
        var count_true = new OmnificentDictionary<CastGroup, int>();
        var count_false = new OmnificentDictionary<CastGroup, int>();
        var count_total = new OmnificentDictionary<CastGroup, int>();
        for (var i = 0; i < assignments.Length; i++)
        {
            if (assignments[i] == true)
                count_true[cast_groups[i]] += 1;
            else if (assignments[i] == false)
                count_false[cast_groups[i]] += 1;
            count_total[cast_groups[i]] += 1;
        }
        foreach (var (cg, total) in count_total)
        {
            int max = (total + 1) / 2;
            if (count_true[cg] > max || count_false[cg] > max)
                return false;
        }
        foreach (var (cg, total) in count_total)
        {
            int min = total / 2;
            if (count_true[cg] < min || count_false[cg] < min)
                return null;
        }
        return true;
    }

    protected override Solver<Applicant> BuildSatSolver(List<(CastGroup,
HashSet<Applicant>)> applicants_needing_alternative_cast)
    {
        var all_applicants = applicants_needing_alternative_cast.SelectMany(p =>
p.Item2).ToArray();
        var applicant_cast_groups = all_applicants.Select(a =>
a.CastGroup!).ToArray();
        return new DpllTheorySolver<Applicant>(soln =>
AlternativeCastsAreEqual(soln.Assignments, applicant_cast_groups), all_applicants);
    }

    protected override Solution FindSatSolution(Solver<Applicant> sat,
List<(CastGroup, HashSet<Applicant>)> applicants_needing_alternative_cast,
```

```
        List<Clause<Applicant>> existing_assignments, List<Clause<Applicant>>
same_cast_clauses, Dictionary<Applicant, SameCastSet> same_cast_lookup)
    {
        Solution solution = Solution.Unsolveable;
        // Initialise() the starting conditions, then Simplify() until the SAT
solver succeeds
        Results.Clear();
        Initialise(applicants_needing_alternative_cast.Select(p => p.Item2.Count),
out int set_size, out int max_sets);
        do
        {
            // compile clauses
            var clauses = new HashSet<Clause<Applicant>>();
            clauses.AddRange(existing_assignments);
            clauses.AddRange(same_cast_clauses);
            bool any_set_clauses = false;
            foreach (var (cg, hs) in applicants_needing_alternative_cast)
                any_set_clauses |= clauses.AddRange(BuildOverlappingSetClauses(hs,
set_size, max_sets, same_cast_lookup));
            if (clauses.Count == 0)
                break; // no clauses to solve
            // run sat solver
            solution = sat.Solve(new(clauses)).FirstOrDefault();
            Results.Add(new Result
            {
                SetSize = set_size,
                MaxSets = max_sets,
                Variables = sat.Variables.Count,
                Clauses = clauses.Count,
                MaxLiterals = clauses.Max(c => c.Literals.Count),
                Solved = !solution.IsUnsolvable
            });
            if (!solution.IsUnsolvable)
                break; // solved
            if (!any_set_clauses)
                break; // we didn't have any sets clauses, if this didn't solve,
nothing will
        } while (Simplify(ref set_size, ref max_sets));
        return solution;
    }

    // This is called to initialise the overlapping set parameters before we start
    // SAT solving. The default implementation is to start by making overlapping
    // sets of 3 across the whole list of applicants. It may be helpful to override
    // this if your Simplify() method uses state which needs to be reset between
    // runs.
    protected virtual void Initialise(IEnumerable<int> applicants_per_cast_group,
out int set_size, out int max_sets)
    {
        set_size = 3;
        max_sets = applicants_per_cast_group.Max() - set_size + 1; // at this
value, the limit will not be hit before all applicants have been placed in sets
    }

    // If the current SAT problem is unsolvable, this is called to modify the
    // overlapping set parameters to make it a simplier SAT problem to solve.
    // If there is no simplier option, return false to stop trying. As a failsafe,
    // if the set parameters fail to create any sets, iteration will terminate.
    // The default implementation reduces the number of sets by 1 each iteration.
    protected virtual bool Simplify(ref int set_size, ref int max_sets)
    {
        max_sets -= 1;
```

```
        return max_sets >= 0; // allow zero in case the basic clauses are solvable
on their own
    }

    // Creates clauses for overlapped sets of applicants, within one cast group,
    // for each primary criteria
    private IEnumerable<Clause<Applicant>>
BuildOverlappingSetClauses(IEnumerable<Applicant> applicants, int set_size, int
max_sets, Dictionary<Applicant, SameCastSet> same_cast_lookup)
    {
        foreach (var criteria in primaryCriterias)
        {
            int set_count = 0;
            var sorted_applicants = applicants.OrderByDescending(a =>
a.MarkFor(criteria)).ToArray();
            for (var i = 0; i < sorted_applicants.Length - set_size + 1; i++)
            {
                if (TakeSet(sorted_applicants, i, set_size, same_cast_lookup) is
Applicant[] set)
                    foreach (var clause in KeepNotAllEqual(set))
                        yield return clause;
                else
                    // TakeSet may fail if the remaining applicants are in a
SameCastSet
                    break;
                set_count++;
                if (set_count == max_sets)
                    break;
            }
        }
    }

    // Creates a clause specifying that not all of the given applicants are in the
    // same alternative cast
    private IEnumerable<Clause<Applicant>> KeepNotAllEqual(Applicant[] applicants);

    // Takes set_size applicants starting at start_index, with at most set_size-1
    // from any one SameCastSet
    private Applicant[]? TakeSet(Applicant[] applicants, int start_index, int
set_size, Dictionary<Applicant, SameCastSet> same_cast_lookup);
}
```

TopPairsSatEngine.cs

```
// A concrete approach for balancing alternative casts by initially pairing off the
// full list of applicants in order, then reducing the number of pairs each
// iteration until success.
// NOTE: this approach may cause uneven alternative cast counts
public class TopPairsSatEngine : PairsSatEngine
{
    public TopPairsSatEngine(IAuditionEngine audition_engine, AlternativeCast[]
alternative_casts, Criteria? cast_number_order_by, ListSortDirection
cast_number_order_direction, Criteria[] criterias);

    protected override bool Simplify(ref int chunk_size, ref int max_chunks)
    {
        max_chunks -= 1;
        return max_chunks >= 0; // allow zero in case the basic clauses are
solvable on their own
    }
}
```

Appendix G - Neural network IAuditionEngine implementations

The relevant functions of the neural network based audition engine implementation is included in this appendix. Although it is not directly analysed in the research, it is included here for completeness.

NeuralAuditionEngine.cs

```
// An AuditionEngine which can learn from the user's choices, by training a
// SingleLayerPerceptron to update the criteria weights.
public class NeuralAuditionEngine : WeightedSumEngine, IComparer<Applicant>
{
    const double MINIMUM_CHANGE = 0.1;

    readonly SingleLayerPerceptron model;
    readonly Criteria[] criterias;
    readonly UserConfirmation confirm;

    // The maximum number of training iterations run per invocation of
    // UserSelectedCast(IEnumerable<Applicant>, IEnumerable<Applicant>)
    public int MaxTrainingIterations { get; set; } = 10;

    // The speed at which the neural network learns from results, as a fraction of
    // MaxOverallAbility. Reasonable values are between 0.001 and 0.01.
    // WARNING: Changing this can have crazy consequences, slower is generally
    // safer but be careful.
    public double NeuralLearningRate { get; set; } = 0.005;

    // Determines which loss function is used when training the neural network.
    public LossFunctionChoice NeuralLossFunction { get; set; } =
LossFunctionChoice.Classification0_3;

    public NeuralAuditionEngine(Criteria[] criterias, UserConfirmation confirm)
        : base(criterias)
    {
        this.criterias = criterias.Where(c => c.Weight != 0).ToArray(); // exclude
criterias with zero weight
        if (this.criterias.Length == 0 && criterias.Length != 0)
        {
            // if all have zero weight, initialise them to equal parts of 100
            this.criterias = criterias;
            var equal_weight = 100.0 / criterias.Length;
            foreach (var criteria in this.criterias)
                criteria.Weight = equal_weight;
        }
        this.confirm = confirm;
        this.model = new SingleLayerPerceptron(this.criterias.Length * 2, 1);
        LoadWeights();
    }

    private void LoadWeights()
    {
        var neuron = model.Layer.Neurons[0];
        neuron.Bias = 0;
        for (var i = 0; i < criterias.Length; i++)
        {
            neuron.Weights[i] = criterias[i].Weight;
            neuron.Weights[i + criterias.Length] = -criterias[i].Weight;
        }
    }
```

```
    public int Compare(Applicant? a, Applicant? b)
    {
        if (a == null || b == null)
            throw new ArgumentNullException();
        var a_better_than_b = model.Predict(InputValues(a, b))[0];
        if (a_better_than_b > 0.5)
            return 1; // A > B
        else if (a_better_than_b < 0.5)
            return -1; // A < B
        else // a_better_than_b == 0.5
            return 0; // A == B
    }

    private double[] InputValues(Applicant a, Applicant b)
    {
        var values = new double[criterias.Length * 2];
        for (var i = 0; i < criterias.Length; i++)
        {
            double max_mark = criterias[i].MaxMark;
            values[i] = a.MarkFor(criterias[i]) / max_mark;
            values[i + criterias.Length] = b.MarkFor(criterias[i]) / max_mark;
        }
        return values;
    }

    public override void UserSelectedCast(IEnumerable<Applicant>
applicants_accepted, IEnumerable<Applicant> applicants_rejected)
    {
        if (criterias.Length == 0)
            return; // nothing to do
        // Generate training data
        var rejected_array = applicants_rejected.ToArray();
        if (rejected_array.Length == 0)
            return; // nothing to do
        var training_pairs = new Dictionary<double[], double[]>();
        foreach (var (accepted, rejected) in ComparablePairs(applicants_accepted,
rejected_array))
        {
            training_pairs.Add(InputValues(accepted, rejected), new[] { 1.0 });
            training_pairs.Add(InputValues(rejected, accepted), new[] { 0.0 });
        }
        if (training_pairs.Count == 0)
            return; // nothing to do
        // Train the model
        model.LearningRate = NeuralLearningRate * MaxOverallAbility;
        model.LossFunction = NeuralLossFunction;
        var trainer = new ModelTrainer(model)
        {
            LossThreshold = 0.005,
            MaxIterations = MaxTrainingIterations,
        };
        _ = trainer.Train(training_pairs.Keys, training_pairs.Values);
        UpdateWeights();
    }

    private void UpdateWeights()
    {
        var neuron = model.Layer.Neurons[0];
        var new_raw = new double[criterias.Length];
        double old_sum = 0;
        double new_sum = 0;
        for (var i = 0; i < criterias.Length; i++)
        {
```

```
            new_sum += new_raw[i] = (neuron.Weights[i] + -neuron.Weights[i +
criterias.Length]) / 2;
            old_sum += criterias[i].Weight;
        }
        var weight_ratio = old_sum / new_sum;
        var new_weights = new double[criterias.Length];
        var any_change = false;
        var msg = "CARMEN's neural network has detected an improvement to the
Criteria weights. Would you like to update them?";
        for (var i = 0; i < criterias.Length; i++)
        {
            new_weights[i] = new_raw[i] * weight_ratio;
            msg += $"\n{criterias[i].Name}: ";
            if (Math.Abs(new_weights[i] - criterias[i].Weight) > MINIMUM_CHANGE)
            {
                msg += $"{new_weights[i]:0.0} (previously
{criterias[i].Weight:0.0})";
                any_change = true;
            }
            else
                msg += $"{criterias[i].Weight:0.0}";
        }
        if (any_change && confirm(msg))
        {
            for (var i = 0; i < criterias.Length; i++)
                criterias[i].Weight = new_weights[i];
            UpdateRange(criterias);
        }
        LoadWeights(); // revert minor or refused changes, update neurons with
normalised weights
    }

    // Finds pairs of good and bad applicants, where the bad applicant is eligible
    // for the cast group of the good
    public static IEnumerable<(Applicant good, Applicant bad)>
ComparablePairs(IEnumerable<Applicant> good_applicants, Applicant[]
bad_applicants);
}
```

Appendix H - Neural network IAllocationEngine implementations

The relevant functions of neural network based allocation engine implementations are included in this appendix. These are compared to the heuristic implementations in Appendix E - Heuristic engine implementations.

ComplexNeuralAllocationEngine.cs

```
// A concrete approach for learning the user's casting choices, using a
// Feed-forward Neural Network with customisable complexity. This complexity
// requires storage of the neural network weights outside the ShowModel.
public class ComplexNeuralAllocationEngine : NeuralAllocationEngine
{
    readonly Lazy<FeedforwardNetwork> model;
    readonly Dictionary<double[], double[]> trainingPairs = new();

    protected override INeuralNetwork Model => model.Value;

    #region Engine parameters
    // If true, training will occur whenever
    // UserPickedCast(IEnumerable<Applicant>, IEnumerable<Applicant>, Role)
    // is called
    public bool TrainImmediately { get; set; } = false;

    // If true, training data will kept to be used again in future training
    public bool StockpileTrainingData { get; set; } = true;

    // If false, the model will only be used for predictions, but not updated
    public bool AllowTraining { get; set; } = true;

    // The method used to save and load the neural network model
    public IDataPersistence ModelPersistence { get; set; }

    // The number of hidden layers to be created in a new model (does not affect
    // loaded models)
    public int NeuralHiddenLayers { get; set; } = 2;

    // The constant number of neurons to be created in a new model layer (does not
    // affect loaded models)
    // NOTE: This is used in conjunction with NeuralLayerNeuronsPerInput
    public int NeuralLayerNeuronsConstant { get; set; } = 0;

    // The number of neurons per input to be created in a new model layer (does
    // not affect loaded models)
    // NOTE: This is used in conjunction with NeuralLayerNeuronsConstant
    public double NeuralLayerNeuronsPerInput { get; set; } = 1;

    // Determines which activation function is used for the hidden layers of a new
    // model (does not affect loaded models)
    public ActivationFunctionChoice NeuralHiddenActivationFunction { get; set; } =
ActivationFunctionChoice.Tanh;

    public override SortAlgorithm SortAlgorithm;
    #endregion

    public ComplexNeuralAllocationEngine(IAuditionEngine audition_engine,
AlternativeCast[] alternative_casts, ShowRoot show_root, Requirement[]
requirements, UserConfirmation confirm, IDataPersistence model_persistence);
```

```
    #region Business logic
    public override void ExportChanges()
    {
        base.ExportChanges();
        if (AllowTraining)
            SaveModelToDisk();
    }

    protected override void AddTrainingPairs(Dictionary<double[], double[]> pairs,
Role role)
    {
        if (!AllowTraining)
            return; // nothing to do
        foreach (var pair in pairs)
            trainingPairs.Add(pair.Key, pair.Value);
        if (!TrainImmediately)
            return; // do it later
        FinaliseTraining();
    }

    protected override void FinaliseTraining()
    {
        if (!trainingPairs.Any())
            return; // nothing to do
        TrainModel(trainingPairs);
        if (!StockpileTrainingData)
            trainingPairs.Clear();
    }

    private FeedforwardNetwork LoadModelFromDisk();

    private FeedforwardNetwork BuildNewModel()
    {
        var neurons_per_layer =
Convert.ToInt32(Math.Ceiling(NeuralLayerNeuronsConstant +
NeuralLayerNeuronsPerInput * nInputs));
        if (NeuralHiddenLayers < 1)
            throw new ApplicationException("The number of hidden layers must be at
least 1");
        if (neurons_per_layer < 2)
            throw new ApplicationException("The number of neurons per layer must be
at least 2");
        var new_model = new FeedforwardNetwork(nInputs, NeuralHiddenLayers,
neurons_per_layer, 1, NeuralHiddenActivationFunction,
ActivationFunctionChoice.Sigmoid); // sigmoid output is between 0 and 1, crossing
at 0.5
        foreach (var neuron in new_model.Layers.First().Neurons)
            FlipPolarities(neuron);
        return new_model;
    }

    private void SaveModelToDisk();
    #endregion

    #region Helper methods
    private void FlipPolarities(Neuron neuron)
    {
        neuron.Bias = 0;
        var offset = nInputs / 2;
        var i = 0;
        foreach (var ow in overallWeightings)
        {
            neuron.Weights[i + offset] *= -1;
```

```
            i++;
        }
        foreach (var requirement in suitabilityRequirements)
        {
            neuron.Weights[i + offset] *= -1;
            i++;
        }
        foreach (var requirement in existingRoleRequirements)
        {
            neuron.Weights[i] *= -1;
            i++;
        }
    }
    #endregion
}
```

NeuralAllocationEngine.cs

```
// The base class of all Neural Network based allocation engines
public abstract class NeuralAllocationEngine : WeightedAverageEngine
{
    protected readonly IOverallWeighting[] overallWeightings;
    private Dictionary<Requirement, int> overallWeightingsLookup;
    protected readonly Requirement[] suitabilityRequirements;
    private Dictionary<Requirement, int> suitabilityRequirementsLookup;
    protected readonly ICriteriaRequirement[] existingRoleRequirements;
    private Dictionary<Requirement, int> existingRoleRequirementsLookup;
    protected readonly int nInputs;
    protected readonly UserConfirmation confirm;

    protected abstract INeuralNetwork Model { get; }

    public Montage? LastTrainingMontage { get; set; } = null;

    #region Engine parameters
    // The maximum number of training iterations run per invocation of
    // UserPickedCast(IEnumerable<Applicant>, IEnumerable<Applicant>, Role)
    public int MaxTrainingIterations { get; set; } = 100;

    // The speed at which the neural network learns from results, as a fraction of
    // the sum of Requirement.SuitabilityWeight. Reasonable values are between
    // 0.001 and 0.01. WARNING: Changing this can have crazy consequences, slower
    // is generally safer but be careful.
    public double NeuralLearningRate { get; set; } = 0.005;

    // Determines which loss function is used when training the neural network.
    public LossFunctionChoice NeuralLossFunction { get; set; } =
LossFunctionChoice.Classification0_4;

    // The sorting algorithm used for ordering the applicants with the neural
    // network
    public virtual SortAlgorithm SortAlgorithm { get; set; } =
SortAlgorithm.OrderBySuitability;
    #endregion

    public NeuralAllocationEngine(IAuditionEngine audition_engine,
AlternativeCast[] alternative_casts, ShowRoot show_root,
        IOverallWeighting[] overall_weightings, Requirement[]
suitability_requirements, ICriteriaRequirement[] existing_role_requirements,
        UserConfirmation confirm);
```

```
    private static Dictionary<Requirement, int> ArrayLookup<T>(int offset, T[]
array);

    #region Business logic
    private bool ConfirmEngineCantLearn();

    public override void UserPickedCast(IEnumerable<Applicant> applicants_picked,
IEnumerable<Applicant> applicants_not_picked, Role role)
    {
        if (role.Requirements.Count(r => suitabilityRequirements.Contains(r)) == 0)
            return; // nothing to do
        // Generate training data
        var not_picked_array = applicants_not_picked.ToArray();
        if (not_picked_array.Length == 0)
            return; // nothing to do
        var training_pairs = new Dictionary<double[], double[]>();
        foreach (var (picked, not_picked) in ComparablePairs(applicants_picked,
not_picked_array))
        {
            training_pairs.Add(InputValues(picked, not_picked, role), new[] { 1.0
});
            training_pairs.Add(InputValues(not_picked, picked, role), new[] { 0.0
});
        }
        if (!training_pairs.Any())
            return; // nothing to do
        // Process training data
        AddTrainingPairs(training_pairs, role);
    }

    // Finds pairs of good and bad applicants with matching cast groups
    public static IEnumerable<(Applicant good, Applicant bad)>
ComparablePairs(IEnumerable<Applicant> good_applicants, IEnumerable<Applicant>
bad_applicants);

    public override void ExportChanges() => FinaliseTraining();

    // Handle the addition of new training pairs, returning suggested weight
    // changes, if any
    protected abstract void AddTrainingPairs(Dictionary<double[], double[]> pairs,
Role role);

    // Handle any remaining training, returning suggested weight changes, if any
    protected abstract void FinaliseTraining();
    #endregion

    #region Neural structure
    private double[] InputValues(Applicant a, Applicant b, Role role)
    {
        var values = new double[nInputs];
        var offset = nInputs / 2;
        double? overall_a = null;
        double? overall_b = null;
        for (var i = 0; i < overallWeightings.Length; i++)
            if (overallWeightings[i] is not Requirement)
            {
                values[i] = overall_a ??= AuditionEngine.OverallSuitability(a);
                values[i + offset] = overall_b ??=
AuditionEngine.OverallSuitability(b);
            }
        foreach (var requirement in role.Requirements)
        {
```

```
            if (overallWeightingsLookup.TryGetValue(requirement, out int
overall_index))
            {
                values[overall_index] = overall_a ??=
AuditionEngine.OverallSuitability(a);
                values[overall_index + offset] = overall_b ??=
AuditionEngine.OverallSuitability(b);
            }
            if (suitabilityRequirementsLookup.TryGetValue(requirement, out var
suitability_index))
            {
                values[suitability_index] = AuditionEngine.SuitabilityOf(a,
requirement);
                values[suitability_index + offset] =
AuditionEngine.SuitabilityOf(b, requirement);
            }
            if (existingRoleRequirementsLookup.TryGetValue(requirement, out var
existing_role_index))
            {
                values[existing_role_index] = CountRoles(a,
((ICriteriaRequirement)requirement).Criteria, role);
                values[existing_role_index + offset] = CountRoles(b,
((ICriteriaRequirement)requirement).Criteria, role);
            }
        }
        return values;
    }

    // Performs a training operation, but doesn't update any weights outside the
    // model
    protected void TrainModel(Dictionary<double[], double[]> pairs)
    {
        Model.LearningRate = CalculateLearningRate();
        Model.LossFunction = NeuralLossFunction;
        var trainer = new ModelTrainer(Model)
        {
            LossThreshold = 0.005,
            MaxIterations = MaxTrainingIterations
        };
        LastTrainingMontage = trainer.Train(pairs.Keys, pairs.Values);
    }

    protected virtual double CalculateLearningRate() => NeuralLearningRate;
    #endregion

    #region Applicant comparison
    public override int Compare(Applicant a, Applicant b, Role for_role)
    {
        if (a == null || b == null)
            throw new ArgumentNullException();
        var a_better_than_b = Model.Predict(InputValues(a, b, for_role))[0];
        if (a_better_than_b > 0.5)
            return 1; // A > B
        else if (a_better_than_b < 0.5)
            return -1; // A < B
        else // a_better_than_b == 0.5
            return 0; // A == B
    }

    protected override List<Applicant> InPreferredOrder(IEnumerable<Applicant>
applicants, Role role, bool reverse = false);
    #endregion
}
```

RoleLearningAllocationEngine.cs

```
// A concrete approach for learning the user's casting choices, by training the
// Neural Network one role at a time, when it is selected.
public class RoleLearningAllocationEngine : SimpleNeuralAllocationEngine
{
    public RoleLearningAllocationEngine(IAuditionEngine audition_engine,
AlternativeCast[] alternative_casts, ShowRoot show_root, Requirement[]
requirements, UserConfirmation confirm);

    protected override void AddTrainingPairs(Dictionary<double[], double[]> pairs,
Role role)
    {
        TrainModel(pairs); // always train immediately
        PropogateChangesToShowModel(role.Requirements.Contains);
    }

    // Empty implementation because training is always processed immediately
    protected override void FinaliseTraining()
    { }
}
```

SessionLearningAllocationEngine.cs

```
// A concrete approach for learning the user's casting choices, by training the
Neural Network with many roles at once,
// at the end of a casting session.
public class SessionLearningAllocationEngine : SimpleNeuralAllocationEngine
{
    readonly Dictionary<double[], double[]> trainingPairs = new();

    #region Engine parameters
    // If true, training will occur whenever
    // UserPickedCast(IEnumerable<Applicant>, IEnumerable<Applicant>, Role)
    // is called
    public bool TrainImmediately { get; set; } = false;

    // If true, training data will kept to be used again in future training
    public bool StockpileTrainingData { get; set; } = true;
    #endregion

    public SessionLearningAllocationEngine(IAuditionEngine audition_engine,
AlternativeCast[] alternative_casts, ShowRoot show_root, Requirement[]
requirements, UserConfirmation confirm);

    #region Business logic
    protected override void AddTrainingPairs(Dictionary<double[], double[]> pairs,
Role role)
    {
        foreach (var pair in pairs)
            trainingPairs.Add(pair.Key, pair.Value);
        if (!TrainImmediately)
            return; // do it later
        FinaliseTraining();
    }

    protected override void FinaliseTraining()
    {
        if (!trainingPairs.Any())
            return; // nothing to do
        TrainModel(trainingPairs);
        if (!StockpileTrainingData)
```

```
            trainingPairs.Clear();
        PropogateChangesToShowModel(r => true);
    }
    #endregion
}
```

SimpleNeuralAllocationEngine.cs

```
// The base class for simple NeuralAllocationEngines, using a
SingleLayerPerceptron, which will store
// their calculated neural network weights entirely within the ShowModel object
structure.
public abstract class SimpleNeuralAllocationEngine : NeuralAllocationEngine
{
    readonly SingleLayerPerceptron model;

    protected override INeuralNetwork Model => model;

    #region Engine parameters
    // Determines when the updated ShowModel weights are reloaded into the neural
    // network.
    public ReloadWeights ReloadWeights { get; set; } =
ReloadWeights.OnlyWhenRefused;
    #endregion

    public SimpleNeuralAllocationEngine(IAuditionEngine audition_engine,
AlternativeCast[] alternative_casts, ShowRoot show_root, Requirement[]
requirements, UserConfirmation confirm)
        : base(audition_engine, alternative_casts, show_root,
                (show_root.CommonOverallWeight.HasValue ? show_root.Yield() :
requirements.OfType<IOverallWeighting>()).Where(ow => ow.OverallWeight !=
0).ToArray(), // zero means disabled
                requirements.Where(r => r.SuitabilityWeight != 0).ToArray(), //
zero means disabled
                requirements.OfType<ICriteriaRequirement>().Where(r =>
r.SuitabilityWeight != 0 && r.ExistingRoleCost != 0).ToArray(), // zero means
disabled
                confirm)
    {
        model = new SingleLayerPerceptron(nInputs, 1); // sigmoid output is between
0 and 1, crossing at 0.5
        LoadWeights();
    }

    #region Neural structure
    private void LoadWeights()
    {
        var neuron = model.Layer.Neurons[0];
        neuron.Bias = 0;
        var offset = nInputs / 2;
        var i = 0;
        double weight_sum = 0;
        foreach (var ow in overallWeightings)
        {
            neuron.Weights[i] = ow.OverallWeight;
            neuron.Weights[i + offset] = -ow.OverallWeight;
            weight_sum += ow.OverallWeight;
            i++;
        }
        foreach (var requirement in suitabilityRequirements)
        {
            neuron.Weights[i] = requirement.SuitabilityWeight;
```

```
            neuron.Weights[i + offset] = -requirement.SuitabilityWeight;
            weight_sum += requirement.SuitabilityWeight;
            i++;
        }
        foreach (var requirement in existingRoleRequirements)
        {
            var weight = CostToWeight(requirement.ExistingRoleCost,
requirement.SuitabilityWeight, weight_sum);
            neuron.Weights[i] = weight;
            neuron.Weights[i + offset] = -weight;
            i++;
        }
    }

    protected void PropogateChangesToShowModel(Func<Requirement, bool> is_relevant)
    {
        var changes = CalculateChanges(is_relevant);
        if (changes.Any())
            UpdateWeights(changes);
    }

    private IEnumerable<IWeightChange> CalculateChanges(Func<Requirement, bool>
is_relevant)
    {
        EnsureCorrectPolarities(model.Layer.Neurons[0]);
        var raw_weights = AverageOfPairedWeights(model.Layer.Neurons[0]);

        var (raw_overall_weights, raw_suitability_weights, raw_role_weights) =
SplitWeights(raw_weights);

        var relevant_weight_ratio = WeightIncreaseFactor(raw_suitability_weights,
raw_overall_weights, is_relevant);
        var total_weight_sum = overallWeightings.Sum(o => o.OverallWeight) +
suitabilityRequirements.Sum(r => r.SuitabilityWeight);

        var normalised_suitability_weights =
NormaliseWeights(raw_suitability_weights, relevant_weight_ratio);
        var normalised_role_weights = NormaliseWeights(raw_role_weights,
relevant_weight_ratio);
        var normalised_overall_weights = NormaliseWeights(raw_overall_weights,
relevant_weight_ratio);

        var new_weights = new Dictionary<ICriteriaRequirement, double>();
        var changes = new List<IWeightChange>();
        for (var i = 0; i < overallWeightings.Length; i++)
        {
            var ow = overallWeightings[i];
            var new_weight = ow is not Requirement requirement ||
is_relevant(requirement) ? normalised_overall_weights[i] : ow.OverallWeight;
            changes.Add(new OverallWeightChange(ow, new_weight));
        }
        for (var i = 0; i < suitabilityRequirements.Length; i++)
        {
            var requirement = suitabilityRequirements[i];
            var new_weight = is_relevant(requirement) ?
normalised_suitability_weights[i] : requirement.SuitabilityWeight;
            if (requirement is ICriteriaRequirement criteria_requirement)
                new_weights.Add(criteria_requirement, new_weight);
            changes.Add(new SuitabilityWeightChange(requirement, new_weight));
        }
        for (var i = 0; i < existingRoleRequirements.Length; i++)
        {
            var requirement = existingRoleRequirements[i];
```

```
            var new_cost = is_relevant((Requirement)requirement) ?
                WeightToCost(normalised_role_weights[i], new_weights[requirement],
total_weight_sum) : requirement.ExistingRoleCost;
            LimitValue(ref new_cost, 0.01, 100);
            changes.Add(new ExistingRoleCostChange(requirement, new_cost));
        }

        return changes;
    }

    private bool UpdateWeights(IEnumerable<IWeightChange> changes)
    {
        bool show_model_updated = false;

        if (changes.Any(c => c.Significant))
        {
            var msg = "CARMEN's neural network has detected an improvement to the
Requirement weights. Would you like to update them?";
            foreach (var change in changes.InOrder())
                msg += "\n" + change.Description;
            if (confirm(msg))
            {
                foreach (var change in changes)
                    change.Accept();
                show_model_updated = true;
            }
            else if (ReloadWeights == ReloadWeights.OnlyWhenRefused)
                LoadWeights(); // revert refused changes
            if (ReloadWeights == ReloadWeights.OnChange)
                LoadWeights(); // revert refused changes, update neurons with
normalised weights
        }
        if (ReloadWeights == ReloadWeights.Always)
            LoadWeights(); // revert minor or refused changes, update neurons with
normalised weights

        return show_model_updated;
    }

    protected override double CalculateLearningRate()
        => NeuralLearningRate * (overallWeightings.Sum(o => o.OverallWeight) +
suitabilityRequirements.Sum(r => r.SuitabilityWeight));
    #endregion

    #region Helper methods
    private static void EnsurePositive(ref double value, double minimum_magnitude =
0.01) => LimitValue(ref value, min: minimum_magnitude);
    private static void EnsureNegative(ref double value, double minimum_magnitude =
0.01) => LimitValue(ref value, max: -minimum_magnitude);
    private static void LimitValue(ref double value, double? min = null, double?
max = null);
    private static double[] NormaliseWeights(double[] raw_weights, double
weight_ratio);
    // Find the average of the matching pairs between the first half of the Neuron
    // weights and the second half. Returned array will have half the length.
    private static double[] AverageOfPairedWeights(Neuron neuron);
    private (double[], double[], double[]) SplitWeights(double[] weights);
    private void EnsureCorrectPolarities(Neuron neuron);
    private double WeightIncreaseFactor(double[] raw_suitability_weights, double[]
raw_overall_weights, Func<Requirement, bool> include_requirement);
    #endregion
}
```

WeightedAverageEngine.cs

```
// A basic AllocationEngine which calculates an Applicant's suitability for a role
// as a weighted average of the suitabilities for each requirement of the role.
public class WeightedAverageEngine : AllocationEngine
{
    protected readonly ShowRoot showRoot;

    public WeightedAverageEngine(IAuditionEngine audition_engine, AlternativeCast[]
alternative_casts, ShowRoot show_root);

    public override double SuitabilityOf(Applicant applicant, Role role)
    {
        var overall_suitability = AuditionEngine.OverallSuitability(applicant); //
between 0 and 1 inclusive
        double score = 0;
        double max = 0;
        foreach (var requirement in role.Requirements)
        {
            score += requirement.SuitabilityWeight *
AuditionEngine.SuitabilityOf(applicant, requirement);
            max += requirement.SuitabilityWeight;
            if (!showRoot.CommonOverallWeight.HasValue)
            {
                score += requirement.OverallWeight * overall_suitability;
                max += requirement.OverallWeight;
            }
        }
        var overall_weight = showRoot.CommonOverallWeight ?? 0;
        if (max == 0 && overall_weight == 0)
            overall_weight = 1; // if no requirements with non-zero weight, we
should apply a non-zero weight to overall
        score += overall_weight * overall_suitability;
        max += overall_weight;
        foreach (var cr in role.Requirements.OfType<ICriteriaRequirement>())
            score -= CostToWeight(cr.ExistingRoleCost, cr.SuitabilityWeight, max) *
CountRoles(applicant, cr.Criteria, role);
        if (score <= 0)
            return 0; // never return a negative suitability
        return score / max;
    }

    // Must be the inverse of WeightToCost(double, double)
    protected double CostToWeight(double cost, double suitability_weight, double
suitability_weight_sum)
        => -cost * (showRoot.WeightExistingRoleCosts ? suitability_weight :
suitability_weight_sum) / 100;

    // Must be the inverse of CostToWeight(double, double)
    protected double WeightToCost(double neuron_weight, double suitability_weight,
double suitability_weight_sum)
        => -neuron_weight / (showRoot.WeightExistingRoleCosts ? suitability_weight
: suitability_weight_sum) * 100;
}
```

Appendix I - Disagreement sort algorithm

In order to be able to handle sorting by inconsistent comparers, such as the comparer resulting from ComplexNeuralAllocationEngine (see section 3.9.4 Complex network allocation model), I created an original sorting algorithm called “Disagreement Sort” based on the classic insertion sort. The pseudocode for the algorithm is included in Figure 56 below, along with the implemented source code as used in CARMEN.

Pseudocode

Input: *a list of items*	***Output:*** *an ordered list of items*

```
function Sort(items)
    initialise an empty list of sets (a set can contain 1 or more items)
    for each item in items
        find the positions within the list where
            - everything in the sets above is less than this item
            - everything in the sets below is greater than this item
        combine all sets between these positions into 1 set
            - if nothing is less, create a new set at the start
            - if nothing is greater, create a new set at the end
            - if there is nothing between, create a new set between them
        insert the item into that set
    if the list only contains 1 set with multiple items
        find the item in the set with the most other items greater than it
        yield this item and remove it from the set # it is the lowest item
    for each set in the list
        if this set contains only 1 item
            yield that item # it is the next lowest item
        else
            yield Sort(set) # recursively sort this set
```

Figure 56 - Pseudocode for the Disagreement Sort algorithm

The basic concept is to perform an insertion sort, but group together any items which do not have a clearly defined position relative to the others. For those items which are grouped, the sort is re-run recursively to separate out any items which are definitely greater than or definitely less than all others in the group. When the sort fails to separate any items, the recursion is terminated and the item with the most other items greater than it is considered to be the lowest and removed from the list. The sort is then run on the remaining items, continuing to separate or remove at least 1 item at a time, until the items are fully sorted.

DisagreementSort.cs

```
// A sort method which works on an imperfect comparison function, that is one where
// A > B > C does not always mean A > C.
public class DisagreementSort<T> : CachedComparer<T>
    where T : class
{
    public DisagreementSort(IComparer<T> imperfect_comparer);

    public IEnumerable<T> Sort(IEnumerable<T> items)
    {
```

```
        var sorted = new List<HashSet<T>>();
        var e = items.GetEnumerator();
        if (!e.MoveNext())
            yield break;
        // First item is always sorted
        sorted.Add(new HashSet<T> { e.Current });
        // Insert each item in order (grouping as required)
        while (e.MoveNext())
        {
            // Find the range which the new item sit in
            int greater_than; // e.Current is greater than everything before (not
including) this index
            for (greater_than = 0; greater_than < sorted.Count; greater_than++)
                if (!GreaterThanAll(e.Current, sorted[greater_than]))
                    break;
            int less_than; // e.Current is less than everything after (not
including) this index
            for (less_than = sorted.Count - 1; less_than >= 0; less_than--)
                if (!LessThanAll(e.Current, sorted[less_than]))
                    break;
            // Insert the new item
            if (greater_than - less_than == 1)
                // The new item has a proper position, insert it there
                sorted.Insert(greater_than, new HashSet<T> { e.Current });
            else if (greater_than <= less_than)
                // Group everything which might be equal to the new item in one
slot
                GroupTogether(sorted, greater_than, less_than, e.Current);
            else
                // Something wacky is going on if this gets hit regularly
                GroupTogether(sorted, less_than, greater_than, e.Current);
        }
        // If only one resulting group, tie breaking is required (to stop infinite
recursion)
        if (sorted.SingleOrDefaultSafe() is HashSet<T> only_group &&
only_group.Count > 1)
        {
            var item_with_the_most_other_items_greater_than_it = only_group
                .OrderByDescending(item => only_group
                    .Count(other_item => other_item != item &&
GreaterThan(other_item, item)))
                .First();
            yield return item_with_the_most_other_items_greater_than_it;
            only_group.Remove(item_with_the_most_other_items_greater_than_it);
        }
        // Enumerate in order, sorting each group recursively
        foreach (var set in sorted)
        {
            if (set.Count == 1)
                yield return set.First();
            else
                foreach (var item in Sort(set))
                    yield return item;
        }
    }

    // Combine the sets at indicies of list between start_index and end_index
    // (inclusive) into one set, and add the new_item.
    private void GroupTogether(List<HashSet<T>> list, int start_index, int
end_index, T new_item)
    {
        // Group together everything which is tied with the new item
        var combined = list[start_index];
```

```
        for (var i = end_index; i > start_index; i--)
        {
            combined.AddRange(list[i]);
            list.RemoveAt(i);
        }
        // Split the group if possible
        var greater_than_items = combined.Where(item => GreaterThan(new_item,
item)).ToHashSet();
        var less_than_items = combined.Where(item => LessThan(new_item,
item)).ToHashSet();
        if (greater_than_items.All(gti => less_than_items.All(lti =>
GreaterThan(gti, lti))))
        {
            list.Insert(start_index + 1, greater_than_items); // after combined
            list.Insert(start_index, less_than_items); // before combined
            combined.RemoveRange(greater_than_items);
            combined.RemoveRange(less_than_items);
        }
        // Add the new item
        combined.Add(new_item);
    }

    private bool GreaterThanAll(T item, IEnumerable<T> greater_than_items)
    {
        foreach (var greater_than_item in greater_than_items)
            if (!GreaterThan(item, greater_than_item))
                return false;
        return true;
    }

    private bool LessThanAll(T item, IEnumerable<T> less_than_items)
    {
        foreach (var less_than_item in less_than_items)
            if (!LessThan(item, less_than_item))
                return false;
        return true;
    }

    private bool GreaterThan(T item, T greater_than_item)
        => Compare(item, greater_than_item) > 0;

    private bool LessThan(T item, T less_than_item)
        => Compare(item, less_than_item) < 0;
}
```

Appendix J - Alternative cast balance results

This appendix contains the full tables of balance cast metrics, as analysed in section 4.1 Comparison of balance between casts.

Abbreviation legend:

HE = Heuristic selection engine
HU = Human produced result
CP = Chunked pairs
TP = Top pairs
HP = Hybrid pairs
Pairs based = average of { CP, TP, HP }
3C = Three's a crowd
RD = Rank difference
BP = Best pairs
Rank based = average of { RD, BP }
NET = average of all results

Mean difference

A lower mean difference is better.

	HE	HU	CP	TP	HP	Pairs based	3C	RD	BP	Rank based	NET
1993	5.1	4.5	1.3	2.7	2.1	2.0	1.8	4.1	2.1	3.1	3.0
1994	3.1	2.0	1.7	2.2	2.3	2.1	2.8	5.4	2.8	4.1	2.8
1995	5.4	3.2	1.1	.9	1.2	1.1	2.8	6.6	2.4	4.5	3.0
1996	3.6	3.8	1.7	1.9	1.8	1.8	3.1	2.6	3.9	3.3	2.8
1997	5.3	2.9	.5	2.0	1.4	1.3	3.0	3.8	3.2	3.5	2.8
1998	5.5	1.4	.7	2.4	2.1	1.7	2.0	2.5	2.0	2.3	2.3
1999	3.4	3.3	.9	2.7	1.0	1.5	2.1	4.1	2.6	3.4	2.5
2000	3.4	3.4	1.0	1.9	3.0	2.0	2.9	7.4	5.2	6.3	3.5
2001	2.9	3.8	1.1	1.3	1.3	1.2	1.4	5.9	3.5	4.7	2.7
2002	4.6	2.8	1.5	1.4	1.1	1.3	1.7	5.6	2.0	3.8	2.6
2003	4.3	2.4	.8	1.5	.6	1.0	1.9	5.4	1.4	3.4	2.3
2004	1.8	2.4	1.5	.9	1.7	1.4	1.7	4.5	3.8	4.2	2.3
2005	2.5	2.1	1.1	1.5	.8	1.1	1.6	2.9	1.5	2.2	1.8

2006	2.5	3.4	.8	1.7	1.1	1.2	1.8	2.6	1.5	2.1	1.9
2007	3.5	2.6	.9	2.3	.8	1.3	1.5	5.1	3.7	4.4	2.6
2008	3.1	1.6	.8	2.3	.8	1.3	2.0	3.0	2.8	2.9	2.1
2009	3.4	2.8	2.2	1.5	.5	1.4	2.0	4.1	1.9	3.0	2.3
2010	3.2	2.4	.9	1.7	1.5	1.4	1.9	2.5	2.7	2.6	2.1
2011	2.1	2.6	.9	2.1	1.4	1.5	1.4	3.6	1.5	2.6	2.0
2012	2.5	3.0	2.1	1.8	1.8	1.9	1.5	4.5	1.7	3.1	2.4
2013	4.1	3.1	.6	3.6	1.5	1.9	1.3	3.0	.9	2.0	2.3
2014	4.7	5.0	.7	1.3	1.3	1.1	1.6	10.3	.8	5.6	3.2
2015	2.1	4.1	2.5	2.0	2.0	2.2	1.8	3.1	2.7	2.9	2.5
2016	5.7	3.9	1.0	1.6	1.9	1.5	2.2	3.8	2.2	3.0	2.8
2017	5.5	2.9	.5	2.4	1.3	1.4	2.7	3.8	2.5	3.2	2.7
2018	7.5	1.3	.6	1.6	.4	.9	1.8	6.4	4.1	5.3	3.0
2019	5.9	4.7	2.6	1.3	1.3	1.7	1.7	2.5	1.3	1.9	2.7
2020	1.8	3.2	1.4	2.3	.8	1.5	1.4	6.0	3.9	5.0	2.6
2021	2.6	3.4	1.5	1.6	1.6	1.6	2.5	4.8	1.6	3.2	2.5

Mean difference (Top 5)

A lower mean difference is better.

	HE	**HU**	**CP**	**TP**	**HP**	**Pairs based**	**3C**	**RD**	**BP**	**Rank based**	**NET**
1993	7.7	5.2	3.0	3.0	3.0	3.0	4.8	9.2	2.7	6.0	4.8
1994	3.7	2.3	4.5	2.2	2.5	3.1	3.3	9.8	4.8	7.3	4.1
1995	6.0	3.8	3.5	2.0	1.7	2.4	5.3	8.7	2.0	5.4	4.1
1996	5.0	3.8	.5	.5	.5	.5	1.8	2.8	.5	1.7	1.9
1997	6.5	3.5	2.7	1.5	1.5	1.9	3.7	6.0	1.8	3.9	3.4
1998	7.6	1.6	2.7	1.1	1.1	1.6	4.6	3.1	1.1	2.1	2.9
1999	6.5	4.7	2.5	1.7	2.0	2.1	4.0	5.7	2.3	4.0	3.7
2000	5.3	4.2	2.2	1.2	1.2	1.5	4.3	6.6	1.1	3.9	3.3
2001	2.3	3.8	1.7	.7	.7	1.0	2.2	8.5	.7	4.6	2.6
2002	5.1	3.6	4.3	3.0	3.0	3.4	2.9	9.8	2.0	5.9	4.2
2003	9.2	3.8	1.8	.8	.8	1.1	3.9	6.9	.8	3.9	3.5

2004	4.3	3.1	3.6	2.4	2.5	2.8	2.9	8.6	2.4	5.5	3.7
2005	3.0	3.0	1.1	1.3	.9	1.1	1.5	4.8	.9	2.9	2.1
2006	2.8	1.5	1.9	2.6	2.6	2.4	3.3	5.1	4.8	5.0	3.1
2007	5.5	2.2	2.3	1.2	1.0	1.5	3.0	7.3	1.0	4.2	2.9
2008	4.8	2.0	2.4	2.1	3.2	2.6	4.1	10.8	1.5	6.2	3.9
2009	5.3	1.8	3.7	1.5	1.5	2.2	5.5	5.0	1.5	3.3	3.2
2010	4.7	1.9	1.5	.9	.9	1.1	2.5	4.7	.7	2.7	2.2
2011	4.2	2.3	2.5	1.5	1.5	1.8	2.8	5.0	1.5	3.3	2.7
2012	2.1	2.9	2.9	.9	.9	1.6	2.9	6.7	.9	3.8	2.5
2013	6.4	4.4	2.3	2.1	1.9	2.1	2.7	4.7	1.9	3.3	3.3
2014	4.4	6.9	2.8	1.7	1.7	2.1	4.2	13.4	1.7	7.6	4.6
2015	4.6	5.7	3.9	1.7	1.7	2.4	4.4	4.9	1.4	3.2	3.5
2016	4.6	3.0	2.8	1.5	1.7	2.0	1.9	6.8	1.2	4.0	2.9
2017	4.0	2.5	2.0	1.7	1.5	1.7	1.7	3.1	2.0	2.6	2.3
2018	9.5	1.9	2.4	2.2	1.4	2.0	1.9	13.3	1.4	7.4	4.3
2019	6.6	4.2	2.4	1.8	1.8	2.0	1.8	4.5	1.2	2.9	3.0
2020	3.7	4.3	3.4	.9	.9	1.7	3.0	6.9	.9	3.9	3.0
2021	4.8	3.2	3.0	2.0	2.0	2.3	3.7	6.2	2.0	4.1	3.4

Rank difference

A lower rank difference is better.

	HE	**HU**	**CP**	**TP**	**HP**	**Pairs based**	**3C**	**RD**	**BP**	**Rank based**	**NET**
1993	17.5	16.7	3.3	10.5	4.8	6.2	11.5	25.5	9.3	17.4	12.4
1994	20.8	21.0	3.8	15.7	9.8	9.8	7.3	22.7	22.7	22.7	15.5
1995	26.0	13.5	5.7	3.0	3.8	4.2	7.2	27.7	5.8	16.8	11.6
1996	21.8	18.3	5.2	10.3	12.5	9.3	13.8	21.7	13.7	17.7	14.7
1997	25.0	14.5	1.7	16.0	6.0	7.9	13.0	18.3	12.3	15.3	13.4
1998	35.8	10.0	3.7	8.3	9.7	7.2	14.5	15.3	11.2	13.3	13.6
1999	20.0	17.5	4.5	12.7	6.0	7.7	8.7	35.0	6.2	20.6	13.8
2000	24.0	21.0	3.8	7.5	13.7	8.3	7.2	34.8	17.5	26.2	16.2
2001	13.3	19.5	5.3	2.2	2.2	3.2	8.2	31.5	9.2	20.4	11.4

2002	27.2	17.8	10.3	12.5	6.0	9.6	13.7	39.2	4.8	22.0	16.4
2003	40.3	20.8	5.0	13.2	4.3	7.5	15.0	58.5	9.5	34.0	20.8
2004	16.8	16.2	4.8	6.2	7.5	6.2	8.5	40.8	10.2	25.5	13.9
2005	9.7	12.5	3.8	8.3	7.8	6.6	6.0	21.5	9.7	15.6	9.9
2006	18.5	8.5	2.3	12.2	4.5	6.3	11.2	11.0	11.3	11.2	9.9
2007	16.2	5.8	3.2	7.0	1.8	4.0	3.5	15.5	12.3	13.9	8.2
2008	18.2	11.0	2.8	8.3	4.0	5.0	16.8	28.0	16.3	22.2	13.2
2009	20.8	17.0	9.2	7.2	3.5	6.6	18.7	26.5	4.7	15.6	13.5
2010	28.7	14.0	3.7	12.2	13.7	9.9	9.8	22.2	3.8	13.0	13.5
2011	18.8	19.8	4.5	8.2	7.2	6.6	7.7	34.0	10.3	22.2	13.8
2012	9.0	30.8	4.3	9.2	9.2	7.6	8.8	22.8	10.0	16.4	13.0
2013	24.8	14.0	1.5	16.2	6.8	8.2	3.5	8.0	3.8	5.9	9.8
2014	17.0	16.7	3.2	3.3	3.3	3.3	4.5	22.2	2.8	12.5	9.1
2015	27.5	24.0	6.2	24.0	24.0	18.1	18.5	57.5	31.5	44.5	26.7
2016	37.8	18.7	4.7	13.8	6.3	8.3	10.8	32.3	20.0	26.2	18.1
2017	37.7	30.7	12.5	23.8	5.7	14.0	21.3	30.5	20.7	25.6	22.9
2018	44.0	9.2	3.2	10.7	3.0	5.6	10.3	99.2	27.2	63.2	25.9
2019	17.3	31.8	5.0	11.7	11.7	9.5	9.3	16.0	20.0	18.0	15.4
2020	13.8	23.3	4.2	13.7	7.5	8.5	5.5	39.3	20.2	29.8	15.9
2021	30.5	27.5	5.0	8.0	8.0	7.0	16.5	29.3	2.8	16.1	16.0

Rank difference (Top 5)

A lower rank difference is better.

	HE	**HU**	**CP**	**TP**	**HP**	**Pairs based**	**3C**	**RD**	**BP**	**Rank based**	**NET**
1993	4.0	4.3	2.8	2.2	2.0	2.3	3.7	5.2	1.7	3.5	3.2
1994	4.8	1.5	3.8	1.7	2.5	2.7	3.3	8.0	4.0	6.0	3.7
1995	6.5	3.7	3.2	1.8	1.3	2.1	5.8	8.2	1.8	5.0	4.0
1996	5.3	4.0	1.0	1.0	1.0	1.0	2.0	2.8	1.0	1.9	2.3
1997	5.8	3.3	3.8	1.8	1.8	2.5	4.5	8.2	2.3	5.3	3.9
1998	7.0	1.7	3.7	1.3	1.5	2.2	5.2	3.2	1.5	2.4	3.1
1999	7.7	4.5	2.8	1.5	2.2	2.2	4.8	5.7	2.2	4.0	3.9

2000	7.8	4.3	2.7	1.5	1.5	1.9	4.7	6.7	1.2	4.0	3.8
2001	3.0	2.8	2.3	1.5	1.5	1.8	2.5	7.0	1.5	4.3	2.8
2002	7.2	4.7	5.3	2.7	2.7	3.6	3.8	8.3	1.8	5.1	4.6
2003	11.2	4.0	3.2	1.0	1.0	1.7	6.8	9.2	1.7	5.5	4.8
2004	5.0	3.8	3.3	2.2	2.0	2.5	3.5	7.5	2.2	4.9	3.7
2005	4.5	5.2	1.5	1.7	.7	1.3	2.7	7.3	.5	3.9	3.0
2006	3.5	2.3	1.5	2.5	2.5	2.2	3.8	5.2	6.2	5.7	3.4
2007	6.0	2.3	2.7	1.7	1.2	1.9	2.8	6.7	1.5	4.1	3.1
2008	5.8	1.5	2.2	1.7	3.2	2.4	5.5	12.7	1.8	7.3	4.3
2009	8.0	2.8	4.2	2.0	2.0	2.7	7.2	6.5	2.0	4.3	4.3
2010	5.2	3.2	1.3	1.3	1.3	1.3	2.5	5.0	.8	2.9	2.6
2011	5.7	3.5	3.2	2.2	2.3	2.6	3.7	7.2	2.3	4.8	3.8
2012	2.7	2.8	3.3	1.8	1.8	2.3	4.3	6.5	1.8	4.2	3.1
2013	8.2	4.3	2.7	2.2	2.0	2.3	3.7	5.2	2.0	3.6	3.8
2014	4.0	7.3	2.8	1.5	1.5	1.9	4.5	9.8	1.5	5.7	4.1
2015	8.3	6.2	4.5	2.2	2.2	3.0	5.8	6.7	1.8	4.3	4.7
2016	7.8	3.3	3.0	1.8	2.2	2.3	2.2	7.3	1.2	4.3	3.6
2017	4.5	4.2	3.3	2.7	1.5	2.5	3.2	3.3	2.0	2.7	3.1
2018	11.8	5.3	2.5	2.8	2.2	2.5	3.0	17.0	2.0	9.5	5.8
2019	7.5	2.8	2.7	2.0	2.0	2.2	2.5	6.5	1.7	4.1	3.5
2020	4.5	3.8	3.2	1.8	1.8	2.3	3.2	11.0	1.8	6.4	3.9
2021	6.8	3.8	3.5	1.3	1.3	2.0	5.3	9.2	1.2	5.2	4.1

Time taken

All times are measured in seconds.

Seconds	CP	TP	HP	3C	RD	BP	NET
1993	.3	.7	20.3	.0	5.1	5.7	4.0
1994	.1	.0	2.3	.0	5.4	5.2	1.6
1995	.1	.0	.1	.0	5.0	.2	.7
1996	.1	.1	.3	.0	5.0	3.5	1.1
1997	.2	.0	.0	.0	5.1	2.0	.9

1998	.1	.0	.1	.1	5.7	5.2	1.4
1999	.2	.0	.1	.0	5.1	5.5	1.4
2000	.1	.0	.1	.0	5.1	7.5	1.6
2001	.0	.0	.1	.2	5.0	1.6	.9
2002	.4	.1	.1	.0	5.1	.6	.8
2003	.0	.0	.0	.0	5.1	.9	.8
2004	.1	.0	.0	.0	.3	.8	.2
2005	.2	.0	.1	.6	13.4	.3	1.8
2006	.1	.0	.1	.0	5.0	5.0	1.3
2007	.1	.0	.5	.0	5.0	10.1	2.0
2008	.0	.1	.1	.0	5.1	12.2	2.2
2009	1.9	.1	.1	.0	5.3	.2	1.0
2010	.1	.0	.0	.0	5.2	.3	.7
2011	.0	.0	.0	.0	5.1	.6	.7
2012	.5	.2	.4	.0	5.0	1.6	1.0
2013	.1	.0	.1	.0	1.3	5.7	.9
2014	.2	.1	.1	.0	5.0	.1	.7
2015	3.8	.1	.1	.1	5.6	6.0	2.0
2016	.7	.1	.1	.0	10.8	7.2	2.4
2017	.5	.3	22.4	.1	5.0	7.9	4.5
2018	.3	.3	1.1	.0	23.5	15.5	5.1
2019	.1	.2	.7	.0	5.0	12.0	2.3
2020	.1	.1	.9	.0	5.1	8.6	1.9
2021	1.7	.1	.2	.0	7.4	5.5	1.9

Appendix K - Recommendation accuracy results

This appendix contains the full data tables of recommendation accuracy metrics, as analysed in section 4.2 Comparison of recommendation accuracy.

Comparison accuracy

Engine Type ∈ {Role Learning, Session Learning} Overall Weight Type ∈ {Common, Vector, None} Subtract Costs From ∈ {Final, Requirement} Count Roles By ∈ {Geometric Mean, Whole Roles} Training Schedule ∈ {Every Role, Per Session, Role Stockpiling} Reload Weights ∈ {On Change, Always, Only When Refused}						Comparison Accuracy
SL	V	R	WR	ER	OWR	87.21%
SL	N	R	WR	ER	OWR	87.21%
SL	N	F	WR	ER	OWR	87.21%
SL	V	F	WR	ER	OWR	87.21%
RL	N	R	WR		OWR	87.20%
RL	V	R	WR		OWR	87.20%
SL	V	R	GM	WR	OWR	87.20%
SL	N	R	GM	ER	OWR	87.20%
SL	N	F	GM	ER	OWR	87.20%
SL	V	F	GM	ER	OWR	87.20%
RL	N	R	GM		OWR	87.20%
RL	V	R	GM		OWR	87.20%
RL	V	F	WR		OWR	87.19%
RL	N	F	WR		OWR	87.19%
RL	N	F	GM		OWR	87.19%
RL	V	F	GM		OWR	87.19%
SL	N	R	WR	PS		87.19%
SL	V	R	WR	PS		87.19%
SL	N	F	WR	PS		87.19%
SL	V	F	WR	PS		87.19%
SL	V	R	GM	PS		87.18%
SL	N	R	GM	PS		87.18%

SL	N	F	GM	PS		87.18%
SL	V	F	GM	PS		87.18%
SL	V	F	GM	RS	OWR	87.03%
SL	N	F	GM	RS	OWR	87.03%
SL	V	R	GM	RS	OWR	87.03%
SL	N	R	GM	RS	OWR	87.03%
SL	V	R	WR	RS	OWR	87.02%
SL	N	R	WR	RS	OWR	87.02%
SL	N	F	WR	RS	OWR	87.02%
SL	V	F	WR	RS	OWR	87.02%
IW	V	F	GM			86.98%
IW	V	F	WR			86.98%
IW	V	R	GM			86.98%
IW	V	R	WR			86.98%
IW	N	F	GM			86.79%
IW	N	F	WR			86.79%
IW	N	R	WR			86.79%
IW	N	R	GM			86.79%
HE	C	R	GM			86.69%
HE	C	R	WR			86.69%

Casting accuracy

Engine Type ∈ {Role Learning, Session Learning} Overall Weight Type ∈ {Common, Vector, None} Subtract Costs From ∈ {Final, Requirement} Count Roles By ∈ {Geometric Mean, Whole Roles} Training Schedule ∈ {Every Role, Per Session, Role Stockpiling}					Exact Applicant	Same Marks	Same Relevant Marks	Similar Marks	Similar Relevant Marks
IW	V	F	WR		26.26%	26.29%	43.65%	26.54%	51.98%
IW	V	F	GM		26.26%	26.29%	43.65%	26.54%	51.98%
IW	V	R	WR		26.26%	26.29%	43.65%	26.54%	51.98%

IW	V	R	GM		26.26%	26.29%	43.65%	26.54%	51.98%
SL	N	F	GM	ER	26.06%	26.16%	43.72%	26.43%	52.20%
SL	V	F	GM	ER	26.06%	26.16%	43.72%	26.43%	52.20%
SL	N	R	GM	ER	26.00%	26.09%	43.75%	26.37%	52.21%
SL	V	R	GM	ER	26.00%	26.09%	43.75%	26.37%	52.21%
RL	N	R	GM		25.99%	26.09%	43.73%	26.37%	52.21%
RL	N	F	GM		25.96%	26.01%	43.63%	26.28%	52.11%
RL	V	F	GM		25.96%	26.01%	43.63%	26.28%	52.11%
HE	C	R	WR		25.88%	25.96%	43.44%	26.22%	52.43%
SL	N	F	GM	PS	25.88%	25.98%	43.66%	26.26%	52.14%
SL	V	F	GM	PS	25.88%	25.98%	43.66%	26.26%	52.14%
SL	N	R	GM	PS	25.83%	25.93%	43.65%	26.21%	52.15%
SL	V	R	GM	PS	25.83%	25.93%	43.65%	26.21%	52.15%
SL	V	F	WR	ER	25.79%	25.82%	43.73%	26.10%	52.33%
SL	N	F	WR	ER	25.79%	25.82%	43.73%	26.10%	52.33%
SL	N	R	WR	ER	25.77%	25.81%	43.74%	26.09%	52.31%
SL	V	R	WR	ER	25.77%	25.81%	43.74%	26.09%	52.31%
RL	N	R	WR		25.75%	25.79%	43.72%	26.07%	52.31%
RL	V	R	WR		25.75%	25.79%	43.72%	26.07%	52.31%
HE	C	R	GM		25.76%	25.84%	43.49%	26.09%	52.32%
IW	N	F	WR		25.89%	25.92%	43.44%	26.15%	51.77%
IW	N	F	GM		25.89%	25.92%	43.44%	26.15%	51.77%
IW	N	R	WR		25.89%	25.92%	43.44%	26.15%	51.77%
IW	N	R	GM		25.89%	25.92%	43.44%	26.15%	51.77%
SL	N	R	GM	RS	25.62%	25.71%	43.66%	25.94%	52.11%
SL	V	R	GM	RS	25.62%	25.71%	43.66%	25.94%	52.11%
SL	N	F	WR	PS	25.59%	25.63%	43.65%	25.91%	52.25%
SL	V	F	WR	PS	25.59%	25.63%	43.65%	25.91%	52.25%
SL	N	F	GM	RS	25.56%	25.66%	43.69%	25.89%	52.11%
SL	V	F	GM	RS	25.56%	25.66%	43.69%	25.89%	52.11%
RL	N	F	WR		25.59%	25.65%	43.56%	25.93%	52.16%

RL	V	F	WR		25.59%	25.65%	43.56%	25.93%	52.16%
SL	N	R	WR	PS	25.56%	25.60%	43.56%	25.88%	52.16%
SL	V	R	WR	PS	25.56%	25.60%	43.56%	25.88%	52.16%
SL	N	R	WR	RS	25.37%	25.41%	43.71%	25.63%	52.25%
SL	V	R	WR	RS	25.37%	25.41%	43.71%	25.63%	52.25%
SL	N	F	WR	RS	25.33%	25.37%	43.64%	25.59%	52.10%
SL	V	F	WR	RS	25.33%	25.37%	43.64%	25.59%	52.10%

Appendix L - Role spread results

This appendix contains the full data table of role spread metrics, as analysed in section 4.3 Comparison of role distribution.

Engine Type ∈ {Role Learning, Session Learning} Overall Weight Type ∈ {Common, Vector, None} Subtract Costs From ∈ {Final, Requirement} Count Roles By ∈ {Geometric Mean, Whole Roles} Training Schedule ∈ {Every Role, Per Session, Role Stockpiling}					Singing Roles	Acting Roles	Dancing Roles	Any Roles
HU			GM		50.14%	34.50%	43.05%	73.13%
HU			WR		50.14%	34.50%	43.05%	73.13%
HE	C	R	GM		47.97%	27.60%	39.72%	66.19%
HE	C	R	WR		47.92%	27.60%	39.72%	66.19%
IW	V	F	WR		47.64%	27.55%	38.78%	67.19%
IW	V	F	GM		47.64%	27.55%	38.78%	67.19%
IW	V	R	WR		47.64%	27.55%	38.78%	67.19%
IW	V	R	GM		47.64%	27.55%	38.78%	67.19%
IW	N	F	WR		47.70%	27.46%	38.52%	67.34%
IW	N	F	GM		47.70%	27.46%	38.52%	67.34%
IW	N	R	WR		47.70%	27.46%	38.52%	67.34%
IW	N	R	GM		47.70%	27.46%	38.52%	67.34%
SL	N	F	WR	RS	46.40%	28.40%	38.20%	67.22%
SL	V	F	WR	RS	46.40%	28.40%	38.20%	67.22%
SL	N	R	WR	RS	46.36%	28.39%	38.24%	67.22%
SL	V	R	WR	RS	46.36%	28.39%	38.24%	67.22%
SL	N	F	GM	RS	46.40%	28.43%	38.10%	67.22%
SL	V	F	GM	RS	46.40%	28.43%	38.10%	67.22%
SL	N	R	GM	RS	46.31%	28.53%	38.05%	67.22%
SL	V	R	GM	RS	46.31%	28.53%	38.05%	67.22%
SL	N	F	WR	ER	46.68%	27.84%	38.15%	67.26%
SL	V	F	WR	ER	46.68%	27.84%	38.15%	67.26%

RL	N	R	WR		46.72%	27.75%	38.15%	67.30%
RL	V	R	WR		46.72%	27.75%	38.15%	67.30%
RL	N	F	WR		46.86%	27.66%	38.11%	67.30%
RL	V	F	WR		46.86%	27.66%	38.11%	67.30%
SL	N	F	WR	PS	46.77%	27.71%	38.15%	67.30%
SL	V	F	WR	PS	46.77%	27.71%	38.15%	67.30%
SL	N	R	WR	ER	46.68%	27.79%	38.20%	67.22%
SL	V	R	WR	ER	46.68%	27.79%	38.20%	67.22%
RL	N	R	GM		46.71%	27.75%	38.10%	67.30%
SL	N	F	GM	ER	46.67%	27.79%	38.10%	67.26%
SL	V	F	GM	ER	46.67%	27.79%	38.10%	67.26%
SL	N	R	WR	PS	46.77%	27.57%	38.15%	67.30%
SL	V	R	WR	PS	46.77%	27.57%	38.15%	67.30%
SL	N	R	GM	ER	46.67%	27.74%	38.14%	67.22%
SL	V	R	GM	ER	46.67%	27.74%	38.14%	67.22%
RL	N	F	GM		46.76%	27.66%	38.06%	67.30%
RL	V	F	GM		46.76%	27.66%	38.06%	67.30%
SL	N	F	GM	PS	46.71%	27.61%	38.10%	67.30%
SL	V	F	GM	PS	46.71%	27.61%	38.10%	67.30%
SL	N	R	GM	PS	46.71%	27.57%	38.10%	67.30%
SL	V	R	GM	PS	46.71%	27.57%	38.10%	67.30%

Appendix M - UAT feedback survey and results

During development, a beta version of CARMEN was released to 5 members of the CGS production team for feedback and UAT. A survey was conducted following their testing for which the full list of questions and responses can be found below.

Survey questions

1. What are 3 things you liked about the software? [Free text]
2. What are 3 things you disliked about the software? [Free text]
3. How important are each of these features to you? [Scale 1-5]
    a. Customisable Audition Criteria
    b. Customisable Show Structure (brackets & acts rather than just a straight list of Items)
    c. Complex Requirements (more than just singer/actor/dancer)
    d. Customisable cast Tags
    e. Easy data entry for Applicants and marks
    f. Storing/viewing Applicant photos
    g. Automatically selecting cast based on past choices
    h. Manual editing of selected cast (including cast numbers, alternative cast, and tags)
    i. Keeping siblings in the same cast
    j. Balancing talent between casts
    k. Easy data entry for Items and Roles
    l. Allowing roles to be in multiple items
    m. Visibility of casting progress while casting roles
    n. Easy to manually pick cast for roles
    o. Automatic casting of a single role based on past choices
    p. Automatic balanced casting of multiple roles
    q. Theming of software to represent your Show/Company
    r. Highlighting of errors/inconsistencies (eg. Item sums don’t add to Bracket sum)
4. How well did each of these features work? [Scale 1-5]
    a. As above
5. Was it easy to find the features you wanted? (and which features were easy to find) [Free text]
6. Were any features hard to find? [Free text]
7. Was anything visually confusing or otherwise lacking? [Free text]
8. Did the software give good casting recommendations? (please give examples) [Free text]
9. Were there any situations where the software gave bad recommendations? (please give examples) [Free text]
10. How would you rate the SPEED of the new software, compared to the old software? [Scale 1-5]
11. How would you rate the INTELLIGENCE of the new software, compared to the old software? [Scale 1-5]

12. How would you rate the EASE OF USE of the new software, compared to the old software? [Scale 1-5]
13. Would you consider using this for another production? (or if you aren't involved in other productions, would you recommend it to their production teams) [Yes/No]
14. Any other feedback you'd like to add? (optional) [Free text]

Survey responses

Q	Responses
1	1. Very quick, 2. Much cleverer in the way it allocates roles, 3. Very flexible in that we can change the way we do things and the program adapts well.
	1. capability in terms of role requirements and tags is quite diverse 2. flow of steps were mostly spot on meaning there was limited times I needed to backtrack before something worked or made sense 3. different decision making algorithms could be good for comparing or even may depend on type of show. e.g. single role story shows or multi-role variety shows
	Easy to read, easy to navigate, really clear
	easy to cast show with simple processes tags with images
	1. The explanation tabs in the right hand corner, 2. The front menu shows previous data set ups you may have worked on 3. Having the flexibility of a show configuration
2	1. the initial setup is tricky but once youve done it you can save it as a default starting database every year; 2. No way of printing out the results - cast lists and casting files, but I know that will come later 3.Some of the error message could be enhanced (I have some suggestions)
	1. Errors need more information given or how to fix, also could be helped with troubleshooting documentation. e.g. "cast numbers assigned 1-63, 4 incomplete" no information on how to resolve this 2. Software crashes when trying to delete a bracket 3. Autofill last role based on total cast in bracket would be good.
	I couldn't work out who the applicants I'd cast wrong were in order to fix them, I couldn't delete all the casting in order to start again, it would crash on me if it didn't like what I was doing. E.g. I tried to delete a bracket I was configuring and it didn't like that and quit on me. It wasn't obvious where to add an applicant's photo, but I worked it out - I LOVE that we see their photo as soon as we click their name! The age totals are completely out of whack - it had 1987 as 32 yo and 2015 as 3 years old..
	nothing to add at this stage

	1. Allocating Roles - if you accidentally choose someone twice in one bracket, it says there is an "error" that applicant has multiple roles, but it does not tell you who or where? 2. Not easy to see photos of cast members 3.				
3a	4	4	5 (Most)	4	5 (Most)
3b	5 (Most)	4	5 (Most)	4	4
3c	4	2	5 (Most)	4	3
3d	4	2	4	5 (Most)	4
3e	4	5 (Most)	5 (Most)	5 (Most)	5 (Most)
3f	5 (Most)	4	5 (Most)	5 (Most)	5 (Most)
3g	5 (Most)	3	4	5 (Most)	3
3h	5 (Most)	4	5 (Most)	5 (Most)	5 (Most)
3i	5 (Most)	4	5 (Most)	4	5 (Most)
3j	5 (Most)	4	5 (Most)	5 (Most)	5 (Most)
3k	4	5 (Most)	5 (Most)	5 (Most)	5 (Most)
3l	5 (Most)	5 (Most)	5 (Most)	5 (Most)	5 (Most)
3m	5 (Most)	2	4	5 (Most)	5 (Most)
3n	5 (Most)	5 (Most)	4	5 (Most)	5 (Most)
3o	5 (Most)	2	4	4	3
3p	5 (Most)	3	4	5 (Most)	4
3q	5 (Most)	3	4	3	4
3r	5 (Most)	5 (Most)	5 (Most)	5 (Most)	5 (Most)
4a	5 (Amazing!)	4	5 (Amazing!)	4	5 (Amazing!)
4b	4	3	5 (Amazing!)	5 (Amazing!)	5 (Amazing!)
4c	5 (Amazing!)	4	4	4	4
4d	5 (Amazing!)	4	4	5 (Amazing!)	4
4e	5 (Amazing!)	1 (Didn't work)	5 (Amazing!)	4	5 (Amazing!)
4f	5 (Amazing!)	2	4	4	2
4g	4	3	5 (Amazing!)	5 (Amazing!)	5 (Amazing!)
4h	5 (Amazing!)	4	4	4	3
4i	5 (Amazing!)	4	5 (Amazing!)	4	5 (Amazing!)

4j	5 (Amazing!)	4	5 (Amazing!)	4	5 (Amazing!)
4k	5 (Amazing!)	2	4	4	5 (Amazing!)
4l	5 (Amazing!)	4	4	4	4
4m	5 (Amazing!)	5 (Amazing!)	5 (Amazing!)	5 (Amazing!)	4
4n	5 (Amazing!)	5 (Amazing!)	5 (Amazing!)	5 (Amazing!)	4
4o	5 (Amazing!)	4	5 (Amazing!)	4	5 (Amazing!)
4p	5 (Amazing!)	3	5 (Amazing!)	4	5 (Amazing!)
4q	5 (Amazing!)	5 (Amazing!)	4	4	5 (Amazing!)
4r	4	3	2	4	4

5	Like any new software it took a bit of playing around to get the feel of what's happening but in the end I got quite good at it. Most things were easy to find or to work out without much effort. Whenever I thought of something I would like to do I generally found a button or a word to click on which did it - so pretty impressive!
	Yes almost everything was easy to find.
	Some where, when it was written and obvious, some were harder, such as I didn't know where to add the applicant's photo, but I worked it out.
	relatively easy, more practice & use will assist
	Yes the features i needed were easy to find, building the show and role allocating
6	The setup is the hardest part to wrap your mind around, but that's because of the flexibility in structure and options that has been built into the software - its a small price to pay for the ability to adapt the process to many different situations and to changing needs. It even opens up some possibilities we have never had the chance to do before like a microphone list!
	I couldn't find how it dealt with ages in requirements. So when selecting cast there was overlap with senior and junior. I wasn't sure if having maximum age for junior and minimum age for senior as the same number would work. If its maximum is 14 I couldn't workout if it took all up to 13.999 or 14.0.
	Where to add a photo, and how to find our which cast members to fix when having an issue with casting that needs to be fixed.
	first time user - some buttons hard to find
	I purposely double casted someone in the same bracket, to see what would happen, but it wont notify who it was. It just says the errors. There was no easy way to distinguish the cast member.
7	The only thing lacking is printouts! Bring on Version 2. Yay!
	When selecting cast, particularly cast with alternate casts the order that you select casts seemed to matter but not be documented. When automatically choosing cast, this worked fine but If I selected for

	example 8 junior boys, I couldn't get it to choose gilbert and sullivan based on my own cast selection. There was also an error given that 4 cast were incomplete and they hadn't been assigned a cast number although when looking at the cast selection screen there were no names down the bottom where the cast with missing numbers would go.				
	I think just making it easy for the dummies - really obvious that you "Click here to add photo" etc. I had an error at first when adding a height - perhaps i you had "cm" prewritten at the end, I would know to only type the number. Also, perhaps each category could have a /10 at the end, so they know not to enter a number higher than 10 or /100 etc.				
	hoping when projected the font and colour contrasts are easy to read!				
	Seemed good				
8	Yes, I was surprised at how well it picked people with the right attributes (singers were alwasy well chosen, dance roles only ever got dance capable people) and how well it balanced cast across multiple roles (so not all the best people get in the first role cast and then the worst get in the second role cast.				
	Yes, based on the requirements set for each role, The software returned the most suitable cast member every time I tried. E.g. Together Dance returned cast in the correct cast (JB,JG,SB,SG) ordered by their dance ability based on the requirement of the role				
	Yes, it definitely chose well based on scores, but where you'd have to input manually is things the system doesn't know such as costume size or personality - as these can determine who from your top runners are more appropriate for specific roles.				
	didn't go to this level				
	Yes, it seemed to follow the fairness rule of allocating roles to cast members, instead of the same top actor getting the parts, it spread them out				
9	The only example I could find was a complex one that even manual casting noften misses - that is ensuring a person from the last item in a previous bracket does not have to be in the first item of the following bracket - often that means manually casting that person into a later item first, then going back and doing the other roles.				
	There were no times where it gave a bad recommendation				
	bad examples are when I hadn't preselected the top roles first, so it gave bad recommendations as they were 'all that was left' - so it is important to pick your main roles first within each bracket.				
	didn't go to this level				
	No				
10	5	5	4	5	5
11	5	4	4	5	5

12	4	4	4	5	4
13	Yes	Yes	Yes	Yes	Yes
14	This software is a major advance on what was possible or even conceivable using the old software. It takes the neural network approach much more deeply into the process and gives far more intelligent answers. much more quickly - and it learns as it goes. Highly commended and I can see this opening up new ones to improve our casting process.				
	The only time I felt the order of steps was out was on the first screen, "alternate cast" could be before "select cast into" as you have to remember to go back and tick alternate cast after you have created them. Can't wait till we can import Applicants from CSV!				
	Can our logo be in colour or does it have to be b&w?				
	well done, fantastic improvement on previous casting program we used! Will be great to use as soon as we can!				
	Top work here. A great step forward. Only other feedback off the top of my head is showing a "read only" summary section, maybe if you clicked "It's Showtime" it produced it?				

Appendix N - Project communication log

A log of key dates and communication between parties involved in this project.

Project Title:		Casting Role Allocation through SAT solving and neural networks		
Student Name:		David Lang	**Supervisor Name:**	Dr Andrew Johnston/ Dr Robert Lang
Date		**Event**	**Topic of Communication**	**Outcome**
24/06/2021	AJ	Email	Initial project kickoff	-
29/06/2021	-	Github	Initial code commit	-
1/07/2021	AJ	Email	ProjectMatch	Confirmed Dr Johnston as thesis supervisor in UTS ProjectMatch.
06/07/2021	RL	Online meeting	Technical requirements/ requested features	Discussed requirements of the application and the history of previous versions.
6/08/2021	AJ	Email	Safety plan	Submitted and signed UTS project Safety Plan.
9/08/2021	AJ	Online meeting	Project plan/ preparatory documents	Good history of project, well written and sound plan, worth identifying the risk of training data for machine learning. Set a secondary goal of user experience to benefit the final product, even if not directly assessed as part of the research. Literature review to be submitted as part of draft in Week 4.
13/08/2021	RL	Online meeting	Testing of v0.1 / letter of involvement	Provided v0.1 prototype release to Rob for testing and feedback (due in 2 weeks). CGS will write a letter of involvement to meet various copyright and data privacy requirements.
16/08/2021	AJ	Online meeting	Prototype application review / amendments to Ethics approval	Reviewed prototype application via Zoom, discussed pros/cons of Desktop/Cloud-based software for this application. Andrew suggested amendments to the Ethics approval form, for David to complete this week.
17/08/2021	RL	Email	Historical casting data	Received historical casting data from CGS, from years 1992-2006 (dos version file format) and 2006-2021 (win version file format).
26/08/2021	AJ	Email	Thesis draft	Submitted draft thesis for review.
28/08/2021	RL	Email	Testing of v0.2	Discussion of issue with v0.1, released v0.2 for testing and further feedback.
28/08/2021	RL	Online meeting	Feedback from testing	Received feedback on v0.1 and v0.2 from testing of prototype application.

2/09/2021	RL	Email	Historical data format	Discussed the file format of the historical casting data, with the aim to create a converter so it can be used for testing.
3/09/2021	AJ	Online meeting	Feedback on draft	Discussed the draft thesis, good background of technical information, layout is clear and logical, methodology is good but could focus more on user feedback at the end.
06/09/2021	RL	Email	Testing of v0.3	Released v0.3 for testing of baseline heuristic engine.
11/09/2021	RL	Email	Feedback from testing	Received feedback on v0.3 from testing of prototype application.
12/09/2021	RL	Email	Testing of v0.4	Released v0.4 for testing of SAT based selection engines.
21/09/2021	AJ	Online meeting	SAT solving results	Discussed SAT solver implementations, the 6 main approaches tested, and the results of testing.
5/10/2021	AJ	Online meeting	Neural network POC	Discussed initial work on coding my own NN, and the likelihood of using the pre-built ML.NET package for proof of concepts.
10/10/2021	RL	Email	Computing resources for model testing	Released software tool to train and evaluate models with various test data and neural network configuration.
19/10/2021	AJ	Online meeting	Neural network evaluation	Discussed preliminary results from (still running) neural network training and evaluation simulations.
19/10/2021	RL	Email	Testing of v0.5	Released v0.5 for testing of neural network based audition and allocation engines.
24/10/2021	RL+	Email	Beta version v0.9 for UAT	Released v0.9 to the beta testers for user acceptance testing and feedback.
28/10/2021	-	GitHub	Release v1.0	MVP released on GitHub with open source under the GPLv3 license.
3/11/2021	AJ	Online meeting	Thesis structure	Neural network training and evaluation simulations are now complete. Discussed progress on thesis writing, and structure.